ENHANCED DEEP REINFORCEMENT LEARNING Q-NETWORK MODELS

BY

ANAS MOHAMMED ALBAGHAJATI

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

DECEMBER 2017

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **ANAS MOHAMMED ALBAGHAJATI** under the direction of his thesis adviser and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER SCIENCE**.

Thesis Committee

Dr. Lahouari Ghouti (Adviser)

Dr. Wasfi G. Al-Khatib (Member)

Dr. Sami Zhioua (Member)

Dr. Khalid A. Al-Jasser
Department Chairman

Dr. Salam A. Zummo
Dean of Graduate Studies

19/2/18
Date

Dedicated to my parents, my brother and my sister

ACKNOWLEDGMENTS

First and foremost praise is to ALLAH, the Almighty, the Greatest of all, on Whom ultimately we depend for sustenance and guidance. I would like to thank Almighty Allah for giving me the opportunity, determination and strength to successfully carry out my research. His continuous grace and mercy have been with me throughout my life and ever more during the tenure of my research.

I would like to express my deep and sincere gratitude to Dr. Lahouari Ghouti, my supervisor, who his continuous support, guidance, help and motivation that encouraged me to accomplish this thesis.

I would like to thank my committee members: Dr. Wasfi Al-Khatib and Dr. Sami Zhioua for their full trust in this work and their valuable constructive comments.

My sincere thanks are also expressed to all my colleagues and friends who shared with me the learning path throughout the graduate studies and supported me in this research with their comments and discussions.

Finally, I would like to express my sincere and heartfelt gratitude to my family who supported and inspired me to strive hard for my ambitious goals, allowing me to enjoy the memorable journey of working through the development of this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

THESIS ABSTRACT

NAME:	Anas Mohammed Albaghajati
TITLE OF STUDY:	Enhanced Deep Reinforcement Learning Q-Network Models
MAJOR FIELD:	Computer Science
DATE OF DEGREE:	December 2017

Artificial Intelligence (AI) is considered one of the hottest and fastest developing areas in Computer Science. Researchers from different disciplines have been trying to understand the human intelligence and transfer such ability to machines. The advancement in technology along with the novelties of proposed algorithms have allowed researchers to achieve breakthroughs throughout the years from defeating the world grandmaster in chess by deep blue to the more recent and ground-breaking defeat of the world champion of the Chinese Go game by a computer. These recent achievements have been made possible with a new technique that merges between two well-established fields in AI and Machine Learning (ML): Reinforcement Learning (RL) and Deep Learning (DL). This astute merger resulted in the fast-evolving field of Deep Reinforcement Learning (DRL). This new set of

algorithms has the ability to use the advancements in DL field to develop feature approximators that aid the learning process of RL using high-dimensional imagery inputs. With such advancements, it became mandatory to have very powerful computing machines for the training of intelligent agents, creating a barrier against researchers and developers with low to mid-range computing power capabilities. Therefore, these restrictions pave the way for developing optimization techniques that reduce the dependency on considerable power without compromising the learning performance. This thesis introduces a preprocessing step that can simplify one of the baseline DRL algorithms commonly known as Deep Q-Network (DQN). This preprocessing step will lead to a simpler model architecture. This architecture combined with the new proposed technique has shown promising results with faster training time and enhanced performance when compared to the original DQN model. Furthermore, the experiments have shown that the proposed technique can be implemented on improved DQN models such as Double DQN (DDQN) and Dueling DQN leading to an improved performance and reduction in training time.

ملخص الرسالة

الاسم أنس محمد البغجاتي

عنوان الرسالة نماذج محسنة من شبكة كيو في التعليم العميق المعزز

التخصص علوم الحاسب الآلي

تاريخ الدرجة العلمية ديسمبر ٢٠١٧

يعتبر الذكاء الاصطناعي أحد أهم وأسرع المجالات النامية في الحاسب الآلي. وقد حاول الباحثون من مختلف التخصصات في فهم الذكاء البشري ونقل هذه القدرة إلى الآلات. وقد أدى التقدم في التكنولوجيا جنباً إلى جنب مع المستجدات في الخوارزميات المقترحة من قبل الباحثين في تحقيق تقدمات ملحوظة على مر السنين ابتداء من هزيمة بطل العالم في الشطرنج من قبل ديب بلو وانتهاء بهزيمة اللاعب الأسطوري العالمي في لعبة جو الصينية مؤخراً من قبل الحاسب الآلي. ولم يكن ليتم تحقيق هذه الإنجازات الأخيرة إلا مع وجود تقنية جديدة تدمج بين اثنتين من المجالات الراسخة في الذكاء الاصطناعي وتعلم الآلة: التعليم المعزز والتعلم العميق. هذا الاندماج الذكي نتج عنه تقنية ذات وتيرة تطور سريعة وهي تقنية التعليم العميق المعزز. هذه المجموعة الجديدة من الخوارزميات لديها القدرة على استخدام التقدم في التعلم العميق لتطوير دوال تقريبية للخصائص والتي بدورها تساعد على عملية التعلم في عملاء التعليم المعزز باستخدام مدخلات الصور عالية الأبعاد. مع هذه الخوارزميات المتقدمة، أصبح لزاماً ومن الأهمية بمكان أن تكون آلات الحوسبة قوية جدا وذات مواصفات تقنية عالية لتكون قادرة على تدريب عملاء بهذا الذكاء الاصطناعي. وقد خلق هذا حاجزا ضد الباحثين والمطورين أصحاب القدرات الحاسوبية منخفضة إلى متوسطة المدى. ولذلك فقد مهدت هذه القيود الطريق لتطوير تقنيات تحسين ورفع كفاءة والتي بدورها تقلل من الاعتماد على القوة الحاسوبية الكبيرة دون المساس بأداء التعلم. هذه الأطروحة تقدم خطوة معالجة استباقية والتي يمكن أن تبسط واحدة من الخوارزميات الأساسية في التعليم العميق المعزز والممثلة بشبكة كيو العميقة مما أدى إلى بناء نموذج أبسط. وقد أظهر هذا النموذج جنبا إلى جنب مع التقنية المقترحة الجديدة نتائج واعدة مع وقت تدريب أقل وتحسن في الأداء مقارنة بالنموذج الأصلي. علاوة على ذلك، فقد أظهرت التجارب قابلية تطبيق التقنية المقترحة على نماذج مطورة من شبكة كيو العميقة كشبكة

كيو العميقة الثنائية وشبكة كيو العميقة المتبارزة مما أدى إلى تحسن أداء هذه النماذج وخفض الوقت المطلوب لتدريبها.

CHAPTER 1

INTRODUCTION

The intelligence of the creatures has been always fascinating and mysterious. This ability to interact with the surrounding environment has always raised many questions for researchers from different disciplines. How can a toddler learn to walk? How can circus animals learn to do certain tricks efficiently and brilliantly? Computer scientists are among those researchers who have been interested in knowing the answers to these questions in addition to many others. Their work focuses on how to transfer such intelligence and way of thinking to machines and computers such that they can interact with the surrounding environment and carry out certain required tasks efficiently without any human intervention. Perhaps, one of the breakthroughs in the field of AI was IBM supercomputer Deep Blue that was capable of defeating the world chess champion and chess grandmaster Garry Kasparov in 1997 [1]. However, despite its remarkable achievement, it was somehow limited. The limitation came from the fact that it was only capable of playing chess without a way of generalizing the results to other tasks. This

was due to the hand-crafted rules that the intelligence was built based on [2]. This limitation motivated the researchers to discover new learning algorithms and techniques that can allow computers to learn such rules without any human intervention. This led the researchers to use RL. RL allows computer agents to learn certain tasks by trial-and-error to achieve the maximum possible reward. This can be related to the way animals learn how to perform the tasks. Its real objective is to actually get the reward for performing the trick correctly regardless of the trick itself. Therefore, they keep adapting their actions and develop the best policy to achieve that. This realization allowed the researchers to achieve great results especially when combined with current advancements in technology. In addition to that, recent state-of-the-art DL algorithms combined with RL have shown great capability in achieving even better results. This combination has been introduced as DRL where the pioneer in this new technique was DeepMind by introducing the first algorithm to combine Convolutional Neural Networks (CNN) with RL as nonlinear function approximator in their DQN technique [3]. Consequently, several researchers since DQN have been proposing a variety of techniques to improve DQN and surpass its results. However, most of the techniques that have been proposed in the literature focus on improving the intelligence of the developed DRL agents. The more sophisticated the algorithms and the neural networks architectures are the more intelligent the agents would be. This is valid because of the features that the agent can extract with these powerful tools. However, this sophistication comes with a price. The more complex the agent is the more

dependant it becomes on processing power. The training of the agents can take several hours that can extend to several days or even over a month. This creates a burden on researchers and developers with low to mid-range machines. Therefore, it introduces a necessity to develop optimized techniques that help developers and researchers in training DRL agents that while maintaining comparable intelligence capabilities to advanced and sophisticated agents, yet they require less time and processing power to train. There have been some researchers who proposed some improvements over DQN without increasing the complexity ([4]) while others have utilized multithreading techniques to increase the efficiency of the algorithms ([5]).

1.1 Problem Statement

The complexity of the algorithms and the sophistication of model architectures while can ensure the development of better and more intelligent DRL agents yet it increases the time required for training such agents. This creates barriers against researchers and gives those with more powerful machines an advantage over the rest. This urges the researchers to develop optimization techniques that can reduce the training time of DRL techniques while maintaining their performance and this would give a chance to all researchers and developers to contribute to the development of the field of DRL regardless of the specifications of their development machines.

1.2 Thesis Contribution

The main contribution of the thesis is the proposal of a new technique that reduces the complexity of the CNN architecture used in DQN to reduce the time necessary for training and at the same time improves the performance of the trained agents. Furthermore, this technique can be applied to any improved and modified versions of DQN such as Double DQN and Dueling DQN resulting in the same optimization behaviour of better performance and less training time.

1.3 Thesis Breakdown

The thesis contains the following chapters. Chapter 2 explains background information about the fields of RL and DL the two essential parts in DRL and it discusses the related work and the recent advancements and state-of-the-art algorithms in DRL. Chapter 3 introduces the proposed technique that can reduce the complexity of DQN algorithm and explains its details while chapter 4 discusses all the conducted experiments to prove the effectiveness of the proposed technique and their results. Finally, chapter 5 concludes the thesis and discusses future work and what can be considered a threat to the validity of the presented work.

CHAPTER 2

BACKGROUND AND LITERATURE REVIEW

2.1 Reinforcement Learning

RL as part of AI is considered to be one of the areas that has been inspired by the nature. In particular, it was inspired by the behaviorist psychology of human beings and animals. This made it the closest learning technique to how humans and animals learn [6]. The essence of intelligent learning is to learn the cause and effect and that is to learn by interacting with the surrounding environment and observing the results and consequences of taken actions. All this governed and motivated by the need to achieve certain goals and collect attractive rewards [7]. RL as a science is interested in finding and discovering methods that solve the problem of learning what behavior to follow in certain situations to achieve maximum reward signal. In the field of ML, RL can be considered a stand-alone paradigm

because of its unique characteristics that distinguish it apart from supervised and unsupervised learning [8]. When compared with unsupervised learning, having an agent that acts in an environment without any supervision regarding what actions to take can be considered to some extent as unsupervised learning. However, RL problems have the reward signal that the agent tries to maximize which acts as a guide for the agent despite the fact that the feedback of this reward signal can be delayed by several timesteps. Nonetheless, the maximization of this signal is in fact considered a totally different objective from the one sought by unsupervised learning where the objective of the latter is to group and categorize unlabeled data in order to find hidden structures. On the other hand, when compared to supervised learning, the difference can be clearer. Supervised learning depends on labeled data for training and the objective is to create prediction models that are capable of classifying or predicting new data. One of the unique characteristics of RL is that it is based on learning by interacting with an environment which is considered closed-loop since the actions taken by the agent will have an impact on the environment even though the consequences of the actions might not appear immediately and be delayed over several timesteps [7]. Thus, the effects of any taken action cannot be completely predicted and therefore monitoring the environment on a regular basis is requested from the agent in order to be able to react accordingly. Another key characteristic of RL is that it considers the problem it tries to solve of an agent trying to achieve certain goals by interacting with an environment that can be uncertain as a whole [7]. This is in contrast with other

paradigms where they can divide the problem into smaller ones and try to solve them without considering the whole picture of the problem.

RL has several elements that can define its system. In addition to the agent and the environment, four other elements can be recognized: an agent policy, a reward signal, a value function and a model of the environment. However, the latter is considered as an optional element. The policy is directly related to the agent. Actually, it is considered as the core of RL agent since it defines how the agent behaves in the environment at a given time and in fact it can alone determine the behavior of the agent sufficiently [7]. This definition of the behavior can be considered roughly to some extent as a mapping between environment's states and actions to be taken by the agent in them. Policies range from simple and naive to more complex and sophisticated ones. They can be as simple as lookup tables or can be more demanding like search processes that request extensive computations. The goal in RL problems can be defined by the second element which is the reward signal. The reward signal helps the agent in distinguishing good from bad events. However, it is limited in a sense that it can only be helpful in a short-term manner. This is because the rewards are immediate and based on current situation representing an intrinsic desirability of the state. However, on the other hand, the value function which is the third element is the one that is more concerned in identifying the good events in a long-term. The value function defines the total cumulative expected reward that the agent will receive starting from the current state and considering the states that are likely to be followed in

the future. Thus, it defines the long-term desirability of being in a state. Because of this, rewards can be considered as primary whereas value functions that predict rewards as secondary. In another sense, without rewards we cannot have values. However, the only reason to have value functions is to achieve more rewards [7]. In fact, when the agent evaluates actions and takes decisions, it relies on the values. Because of the importance of value functions, it is considered the most important component that differentiates between RL algorithms [8]. The fourth and last element of RL systems that is sometimes considered as an optional one is a model of the environment. The objective of the model is to mimic the behavior of the environment. This gives the ability to infer the behavior of the environment. However, not all methods and algorithms in RL depend on environment models. Those methods that use models for solving RL problems are called model-based methods and considered as planning algorithms [7]. On the other hand, there are the simpler model-free methods. These methods are considered to be trial-and-error learners which are viewed as almost the opposite of planning [7]. If RL systems were compared to biological ones, it would be thought of policy as set of stimulus-response rules whereas the rewards are similar to the experiences of pleasure or pain [9].

RL has its share of challenges. In fact, one of the well-known challenges that appears in RL but not in other paradigms is the exploration-exploitation trade-off [10]. The definition of both terms can be understood from the context of RL problem. In RL, the agent tries to learn the best policy that can help it in

achieving the maximum reward. However, while developing the policy, the agent will be confronted to follow one of two scenarios. The first scenario dictates that the agent must utilize the knowledge that it already knows about the actions and their rewards and therefore pick the one with the maximum expected reward. This is called exploitation because the agent keeps utilizing its knowledge. On the other hand, exploration scenario encourages the agent to try new actions in a hope for finding new states that might have better rewards even though those actions may not have the best rewards. In other words, when the agent is exploiting its knowledge to choose the action with the highest reward, it is called to be choosing the greedy action and when exploring new actions, it is called to pick non-greedy ones. Exploitation can be considered rewarding on a short-term whereas exploration is more concerned with maximizing the reward in the long-term [7]. From their definitions, it can be clear that exploration and exploitation cannot occur at the same time and this creates a conflict. The preference of choosing one over the other depends on several factors in a complex way. This includes the precision of values estimation, the uncertainty of the environment and the remaining number of steps before reaching a goal state [7]. Following a greedy method that dictates to choose the greedy action all the time or exploring new action throughout the learning process leads to sub-optimal behavior and this in turn creates the urge for a method to be used in order to balance between the two.

2.1.1 Markov Decision Processes

Markov Decision Processes (MDPs) can be defined as controlled stochastic processes. Those processes satisfy the Markov property [11]. In these processes, a set of states and actions are used to model the environment. Those actions are used to transit from one state to another where the aim is to control the system such that it maximizes certain performance criteria [8]. Satisfying the Markov property insures that the next state to be reached is dependent on the current state and the action taken in that state without any information about the past [7]. A Markov Decision Process can be formally represented as the following [8]

$$\big(S, A, P_{SA}, \gamma, R\big)$$

where :

- S is a set of **states**.
- A is a set of **actions**.
- P_{sa} are the state transition probabilities. Being in state s $\in$ S and taking action a $\in$ A, P_{sa} gives the probability distribution over possible next states. It is also sometimes referred to in literature as Transition Function [10], which is defined as the following T: S $\times$ A $\times$ S' $\rightarrow$ $\big[0,1\big]$. That is the probability of reaching state s' from state s by taking action a denoted as $\mathrm{T}\big(\mathrm{s,a,s'}\big)$ such that $\sum_{s' \in S} T(s, a, s') = 1$.
- γ is called **the discount factor** and takes a value $\in$ $\big[0,1\big)$.

- R is the **reward function**. The reward can be a function of state S and action A $\big($R: S $\times$ A $\mapsto \mathbb{R}\big)$ or state S only $\big($R: S $\mapsto \mathbb{R}\big)$. Furthermore it can be defined in terms of transitions between states as the following: $\big($R: S $\times$ A $\times$ S $\mapsto \mathbb{R}\big)$. Its importance comes for specifying the goal of learning. The value of the reward can be positive for winning states as a reward for reaching those states or negative as penalties for reaching a losing state. Furthermore, non-goal states can be assigned positive rewards that can be considered as sub-goals. All this can define the direction for the agent to learn how to behave.

Once the MDP is defined as $\big($S, A,P_{sa}, γ, R$\big)$, a need emerges to identify how to control the agent in the environment such that each state s $\in$ S is mapped to an action a $\in$ A which is called a **policy**. The policy can be either deterministic or stochastic. A deterministic policy π is a function defined as the following: π: S $\rightarrow$ A. A given policy π can be applied to an MDP as the following: starting in initial state s_0. Then by consulting the policy π, the action a_0 that is mapped to state s_0 ($a_0 = \pi(s_0)$) will be performed. This will lead to a transition to state s_1 with probability T(s_0,a_0,s_1) and a reward r_0 = R (s_0,a_0) will be given. All this is based on the transition function T and the reward function R. Once a transition occurs to s_1 the policy again will be consulted for an action that will lead to another state. The process continues until it reaches a goal state s_{goal} (s_0,a_0,$r_0 \rightarrow s_1$,a_1,$r_1 \rightarrow \cdots \rightarrow s_{goal}$,$a_{goal}$,$r_{goal}$). By this a sequence of states will be visited by taking a sequence of actions. This will lead to achieve cumulative

reward that is equal to the following: $R(s_0,a_0) + \gamma\ R(s_1,a_1) + \gamma^2\ R(s_2,a_2) + \cdots + \gamma^n\ R(s_{goal},a_{goal})$. This formula of the cumulative reward is discounted at each step by a factor of γ. This indicates that the positive rewards received in future steps are taken into consideration. However, they are discounted by the number of steps needed to be taken in order to receive them. This will force the agent to focus on early rewards instead of later ones even if they are bigger. Furthermore, the use of the discount factor ensures that the sum of the rewards is bounded even if the horizon is infinite which means that the number of steps taken is unlimited. Hence, this discounted reward model is called infinite-horizon model that allows the agent to behave optimally [11, 8, 7]. Figure 2.1 shows on the left an example of an environment where the initial state is denoted by the green square and the goal state is denoted by +1. The agent is required to develop an optimal policy that allows it to reach the goal state and at the same time avoid the -1 state. On the right side of Figure 2.1, the policy can be seen as the arrows that appear in each state indicating the optimal action that the agent can take in that state in order to reach the maximum reward.

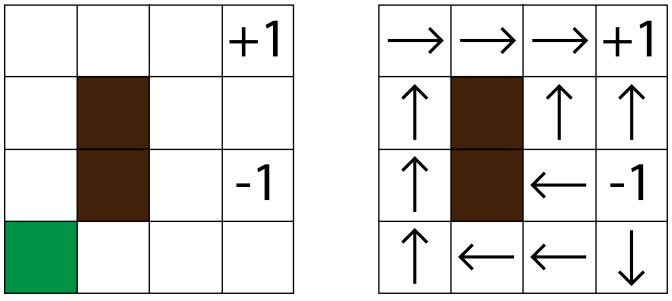


Figure 2.1: Example of Markov Decision Process

Bellman Equations

In order for the agent to behave optimally it should learn optimally. With that said, the agent then must maximize the result of the sequence of its behavior. Therefore, the agent must maximize the expected reward achieved. The expected value of the sum of rewards defined in the previous section is called the **value function**. Given a policy π that maps states S to actions A, a value function for policy π can be defined as the following: $V^{\pi}(\mathrm{s}) = \mathrm{E}\big[\ \mathrm{R}(s_0,a_0) + \gamma\ \mathrm{R}(s_1,a_1) + \gamma^2\ \mathrm{R}(s_2,a_2) + \cdots \mid s_0 = \mathrm{s},\ \pi\ \big]$. This means that $V^{\pi}(\mathrm{s})$ is equal to the expected value of the discounted rewards sum where the state is s and the actions are taken according to policy π. Using the fixed policy π that is given, it is said that $V^{\pi}(\mathrm{s})$ satisfies the **Bellman Equations**,

$$V^{\pi}(s) = R(s,\pi(s)) + \sum_{s' \in S} \gamma T\big(s,\pi(s),s'\big) V^{\pi}(s') \tag{2.1}$$

The format of the Bellman equations can differ based on how the reward function is calculated. The previous format uses a reward function that is based on the state and the action taken. If the reward function is based on the state only, then the Bellman equations will be further simplified to be,

$$V^{\pi}(s) = R(s) + \sum_{s' \in S} \gamma T\big(s,\pi(s),s'\big) V^{\pi}(s') \tag{2.2}$$

However, if the reward function depends on the current state, the action taken and the next state, then the format of Bellman equations will be,

$$V^{\pi}(s) = \sum_{s' \in S} T\big(s, \pi(s), s'\big)\Big(R(s, \pi(s), s') + \gamma V^{\pi}(s')\Big) \tag{2.3}$$

The Bellman equations can be used to solve for V^{π} efficiently. This is because for each state in S, a Bellman equation can be written for $V^{\pi}(s)$. Those equations represent a set of |S| linear equations in |S| variables that can be solved for the unknown values $V^{\pi}(s)$ for each state in S.

The ultimate goal for a given MDP is to find the best policy that gets the most reward. This means finding the maximum value function of Bellman equations of all states s ∈ S. This best policy is called an **optimal policy** that is denoted by π^* such that $V^{\pi^*}(s) \geq V^{\pi}(s)$ for all states s ∈ S and all possible policies π. The value function for the optimal policy can be written in the following format,

$$V^{*}(s) = \max_{a \in A}\Big(R(s, a) + \sum_{s' \in S} \gamma T\big(s, a, s'\big) V^{\pi}(s')\Big) \tag{2.4}$$

Equation (2.4) is called Bellman optimality equation. It shows that under an optimal policy, the value of a state is equal to the expected reward for the best action possible in that state. In order to identify the optimal action for a given optimal value function for a state V^* (and hence the optimal policy) the following function can be used:

$$\pi^*(s) = arg\max_{a \in A} \left(R(s,a) + \sum_{s' \in S} \gamma T\big(s,a,s'\big) V^*(s') \right) \tag{2.5}$$

An interesting property of π^* is that it is optimal to all states s $\in$ S. This means whatever the initial state of the MDP is, the policy will insure that the agent will behave optimally[11, 8, 7].

2.1.2 Value and Policy Iteration

In order to solve an MDP and calculate the optimal values for each state or the optimal policy, there are two standard algorithms that can be used:**Value Iteration and Policy Iteration**.

Value Iteration algorithm can be described as a repeatedly applied algorithm to update the value functions of all states s $\in$ S. This repetition continues until the convergence of the values. This algorithm has two different versions. It can be **synchronous** or **asynchronous**. The synchronous version computes the new values for all states then after that updates the old ones before moving to the next iteration. On the other hand, the asynchronous version iterates over the states and updates the values one at a time within the same loop iteration.

Algorithm 1: Value Iteration Pseudo-Code

Initialize V arbitrarily, *e.g.*, $V(s) = 0$, $\forall s \in S$;

repeat

  $\Delta \leftarrow 0$;

  foreach $s \in S$ **do**

    $v \leftarrow V(s)$;

    $V(s) \leftarrow max_a(\mathcal{R}(s,a) + \sum_{s'}[\gamma \mathcal{T}(s,a,s')V(s')])$;

    $\Delta \leftarrow max(\Delta, |v - V(s)|)$

  end

until $\Delta < \epsilon$*(a small positive number)*;

The second standard algorithm for solving an MDP is **Policy Iteration**. This algorithm starts with initializing the policy π randomly. After that it will execute a loop until convergence where for each iteration of the loop the algorithm will compute the maximum value function over possible actions for each state and update the policy based on that[8].

Algorithm 2: Policy Iteration Pseudo-Code

Initialize $V(s) \in \Re$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily $\forall s \in S$;

1. Policy Evaluation:

repeat

$\Delta \leftarrow 0$;

foreach $s \in S$ **do**

$v \leftarrow V(s)$;

$V(s) \leftarrow \mathcal{R}(s, \pi(s)) + \sum_{s'}[\gamma \mathcal{T}(s, \pi(s), s')V(s')])$;

$\Delta \leftarrow max(\Delta, |v - V(s)|)$

end

until $\Delta < \epsilon$*(a small positive number)*;

2. Policy Improvement:

policy-stable $\leftarrow$ *true* ;

foreach $s \in S$ **do**

$b \leftarrow \pi(s)$;

$\pi(s) = argmax_a(\mathcal{R}(s, a) + \sum_{s'}[\gamma \mathcal{T}(s, a, s')V(s')])$;

if $b \neq \pi(s)$ **then**

policy-stable $\leftarrow false$;

end

end

if *policy-stable* **then**

stop;

else

go to Policy Evaluation

end

2.1.3 Q-Learning

So far it has been assumed that the transition function probabilities and the rewards are given and known. However, it is not the case in real problems. This type of problems can be solved using RL. RL adds on MDP the way to approximate and compute missing information in terms of transition probabilities or rewards. Therefore, the methods to solve for RL can be model-free methods in contrast with the previously discussed algorithms where they depend on the model of MDP. In order to solve an RL problem, Temporal Deference (TD) Learning algorithms can be used. The aim is to estimate the values based on other estimations because of the absence of some information. Each iteration in the learning process generates a learning example that can be used to estimate the values according to the estimated reward in the value of the next state to be in. One of TD algorithms is Q-Learning. The algorithm is aimed to estimate the Q-values $Q(s, a)$ which are the expected discounted total reward starting from state s after taking action a and following policy π. The idea of the algorithm is to incrementally estimate the Q-values for actions based on the received reward and the agent's Q-value function.

$$Q^{\pi}(s_t, a_t) = r_t + \max_{a' \in A} \left(Q^{\pi}(s_{t+1}, a') \right) \tag{2.6}$$

The update function for the Q-values can be represented as the following,

$$Q_{k+1}(s_t, a_t) = Q_k(s_t, a_t) + \alpha\left(r_t + \gamma \max_{a \in A}\left(Q_k(s_{t+1}, a)\right) - Q_k(s_t, a_t)\right) \tag{2.7}$$

where α is called the learning rate that can have a value $\in$ [0,1) and its purpose is to restrict how much information is used in the update of the Q-values [11, 7].

Algorithm 3: Q-Learning Pseudo-Code

Initialize $Q(s, a)$ arbitrarily $\forall s \in S$, $a \in A$ except for

$Q(terminal - state, .) = 0$;

for *episode* $k \leftarrow 1$ to $\mathcal{M}$ do

Initialize S ;

for $t \leftarrow 1$ **to** $Terminal$ **do**

Choose a valid action a_t from state s_t using policy derived from Q_k ;

Take action a_t and observe the reward r_t and the next state s_{t+1};

$Q_{k+1}(s_t, a_t) = Q_k(s_t, a_t) + \alpha\left(r_t + \gamma \max_{a \in A}\left(Q_k(s_{t+1}, a)\right) - Q_k(s_t, a_t)\right)$;

$s_t \leftarrow s_{t+1}$

end

end

Continuous MDPs

Real world problems are further different from what have been discussed previously in modeling MDP problems. The states of those problem are continuous. The solutions that have been proposed previously are all assume that the states are finite. Therefore, in order to handle real world problems, two different solutions can be used. The first one which is the simplest is to discretize the state space of a continuous-state problem. After that, the previously discussed algorithms can be simply used. However, this method suffers from what is called **curse of dimensionality**. This occurs when the states are n-dimensional (n different variable in the state). Assuming that each dimension will be discretized to k different values. Then the result will be k^n different discrete states and this number grows exponentially with respect to the dimension n [8].

2.1.4 Value Function Approximation

The second way of finding the optimal policy for a continuous RL problem is to approximate the value function. In order to achieve that, there must be a simulator for the states. The aim of this simulator is to work as a black-box such that it takes as an input any continuous value state s_t and an action a_t and returns as an output an estimated Q-value [8]. The importance of such approximators can be realized by the fact that during training session, it can be impossible for the agent to visit each and every state in state space especially in complex problems. Therefore, during testing it must rely on value function approximators to predict

a very close Q-values to the real ones. Such approximators can be linear and therefore simple or more complex non-linear ones. In problems such as games which heavily depend on high-dimensional sensory outputs that appear as 2D or 3D graphics through the screen, they definitely require highly complex non-linear approximators. Over the last decade, DL algorithms have proven to be the best in dealing with such high-dimensional sensory inputs to extract meaningful features. By using such approximators, RL algorithms can gain a huge boost in their performance when they learn how to operate optimally in such environments. By using the high-dimensional images that are taken from the games environments and use them as inputs to deep neural networks, those networks can adapt and learn how to approximate the value function of the different actions that can be taken by the agent in a variety of states. Using the values of those actions, the agent can learn how to act optimally in those environments.

By using a neural network as value function approximator, Q-Value will be parametrized over θ to become $Q(s_t, a_t; \theta_i)$. Training the approximator can be achieved by minimizing a sequence of loss functions $L_i(\theta_i)$ that is equal to:

$$L_i(\theta_i) = E_{s,a\sim\rho(s,a)}\Big[(y_i - Q(s, a; \theta_i))^2\Big] \tag{2.8}$$

where

$$y_i = E_{s'\sim\varepsilon}\Big[R(s) + \gamma \max_{a'}(Q(s', a'; \theta_{i-1}|s, a)\Big] \tag{2.9}$$

is considered the target value for iteration i. $\rho(s, a)$ is called the behaviour distribution which is a probability distribution over sequences s and actions a which indicates how probable the agent will end up in state s by taking action a. In real life applications this value cannot be 1 due to uncontrolled errors that might occur. It is worth noting that this approach differs from supervised learning in that the targets change at each iteration depending on the parameters θ_{i-1} where as the targets are fixed before training in supervised learning.

2.2 Deep Learning

In order to understand DL algorithms, it is essential to understand its origins. Inspired by human brains and its base unit the neuron, researchers tried to mimic and transfer such brains to computer machines. The objective was to create adaptive learning systems that have the ability to learn how to perform certain computational tasks including pattern recognition and classification.

2.2.1 Perceptrons

Perceptron was in fact the first neural network algorithm in its simplest forms. It was invented by Rosenblatt in 1958 [12]. The aim was to use this algorithm to linearly separate data using supervised learning. By giving it an input vector of data sample (i) $X^{(i)}$ of m binary variables (denoted as ${x_j}^{(i)}$, where $j = 1, 2, \cdots, m$ and $i = 1, 2, \cdots, n$), it should determine to which of two classes it belongs. The output of a perceptron can be either -1 or 1 corresponding to the positive and negative classes. In order to come up with such decision, a certain mathematical operation is done on the input vector that would give the perceptron the ability to classify the input that can be seen in equation (2.10). What happens is that each input variable (${x_j}^{(i)}$) is multiplied by a weight (w_j) and then they are all summed up together. The result of the summation then goes into a hard-limiter function. If the value is above a threshold then the output is one. Otherwise the output is negative one [13]. These weights (W) can be thought

of as a measurement of the importance of the corresponding input variables to the output [14]. In some implementations, the threshold is sometimes moved to the other side of the equation to become a bias (b). From its name, the value of the bias can affect the decision of the perceptron on leaning towards one class or another. It can in fact shift the decision boundary to one side or another. The latter can be explained if the summation of the inputs and weights multiplication is thought of as a dot product between two vectors [15]. This would result in an equation of a hyperplane that would separate the two classes of the data as it can be shown in Figure 2.3 and in fact the value of the bias b can be embedded in the weights vector as w_0 and its corresponding x value is ${x_0}^{(i)}$ which is equal to 1 and shared among all data samples as it can be seen in equation (2.11).

$$\textbf{output } y^{(i)} = \begin{cases} 1 & \text{if } \sum_{j=1}^{m} w_j {x_j}^{(i)} \geq \text{Threshold} \\ -1 & otherwise \end{cases} \tag{2.10}$$

$$\textbf{output } y^{(i)} = \begin{cases} 1 & \text{if } \sum_{j=1}^{m} w_j {x_j}^{(i)} + b \times 1 = \sum_{j=0}^{m} w_j {x_j}^{(i)} = W^T X^{(i)} \geq 0 \\ -1 & otherwise \end{cases} \tag{2.11}$$

The learning process or what is called training, in a perceptron can be achieved by following a simple learning rule. Starting with a dataset of n samples of X vectors (denoted as $X^{(i)}$, where $i = 1, 2, \cdots, n$) and their linearly separable target classes

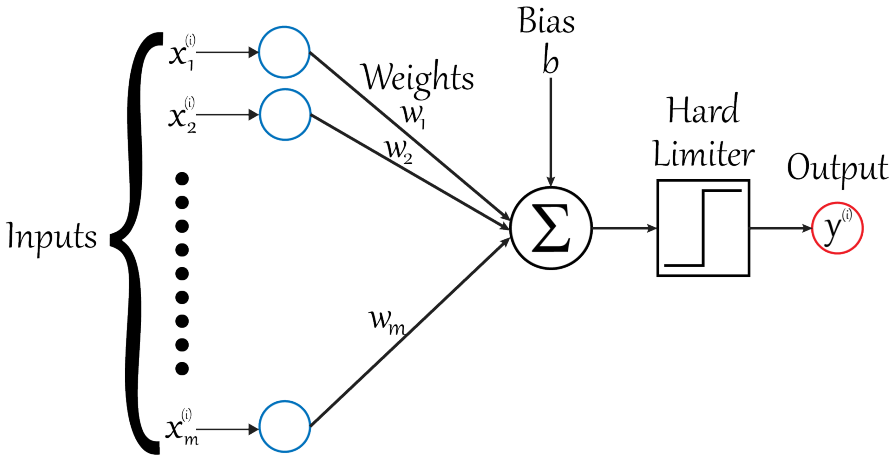


Figure 2.2: Perceptron

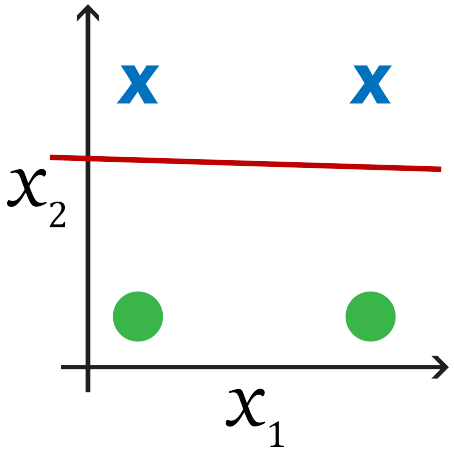


Figure 2.3: Perceptron Hyperplane

(denoted as $t^{(i)}$, where $t = 1$ or -1 and $i = 1, 2, \cdots, n$), then randomly initializing the weights (W) to small numbers, the predicted output can be calculated for one sample at a time using equation (2.11). After calculating the output of each sample, the weights need to be updated based on equation (2.12). The value that is added to the weights in each update (equation (2.13)) is calculated according to how much different the predicted output value from the real target one multiplied by the value of x of that corresponding weight and the learning rate α that takes

a value between 0 and 1 [15]. The random initialization of the weights can in fact create a misplaced hyperplane boundary that would result in misclassifying the inputs as shown in Figure 2.4a. However, with each iteration of training, the update of the weights can modify the hyperplane towards a correct classifier as shown in Figures 2.4b and 2.4c respectively. The bias is also updated in each iteration. However, its update rule is a special case of equation (2.13) where the value of x is equal to 1.

$$w_j = w_j + \Delta w_j, \text{for } j = 1, 2, \cdots, m \tag{2.12}$$

$$\Delta w_j = \alpha(t^{(i)} - y^{(i)}){x_j}^{(i)}, \text{for } j = 1, 2, \cdots, m \tag{2.13}$$

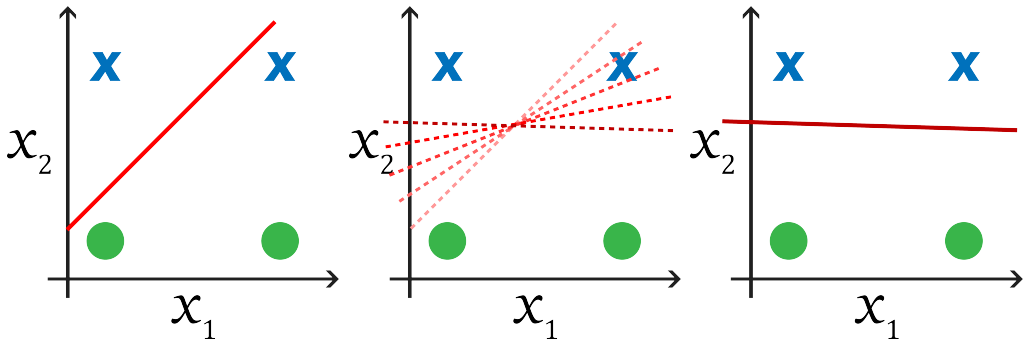


Figure 2.4: Perceptron Learning Process

One of the drawbacks of perceptrons beside the limitation to linearly separable data that will be addressed later, is that one change in the weights based on one data sample can in fact impact the accuracy of the perceptron [15]. The aim of developing learning algorithm is to optimally be able to reflect small changes in

the weights or bias in such a way the learning algorithm will be able to behave correctly. Because of the poor behaviour that perceptrons tend to follow with updates, an update to the algorithm was needed to overcome this issue.

2.2.2 Sigmoid Neurons

A sigmoid neuron is very similar to perceptrons in general. However, it defers in the way it reacts to changes in the weights and bias such that small changes result in small changes in the output [13]. In terms of general structure, it looks similar to a perceptron. However, it accepts as input real values instead of just binary ones. It has a weight associated with each input variable in addition to a bias as in the perceptron. However, the output is no longer produced based on thresholding the result of the summation of the weights multiplied by the inputs added to the bias (equation (2.14)) in order to classify the data sample into positive or negative class. It rather passes the output of equation (2.14) to a sigmoid function $\sigma()$ that is defined by equation (2.15)

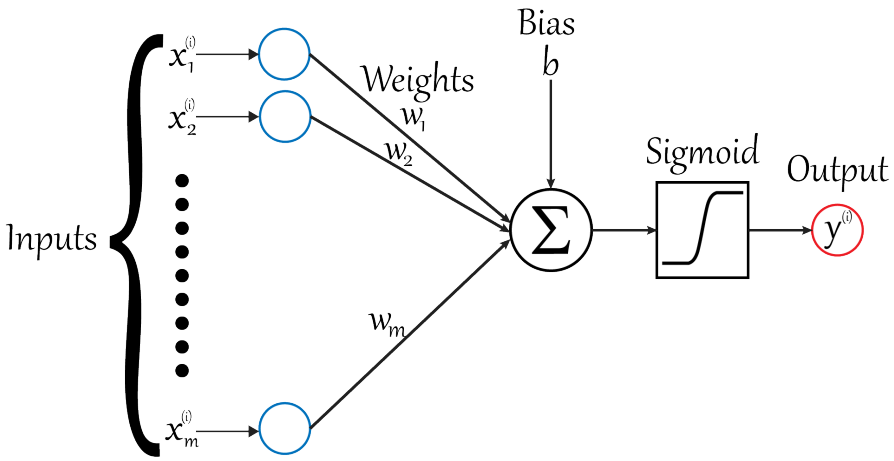


Figure 2.5: Sigmoid Neuron

$$\sum_{j=1}^{m} w_j {x_j}^{(i)} + b \times 1 = \sum_{j=0}^{m} w_j {x_j}^{(i)} = W^T X^{(i)} \tag{2.14}$$

$$\sigma(z) = \frac{1}{1 + \exp(-z)} \tag{2.15}$$

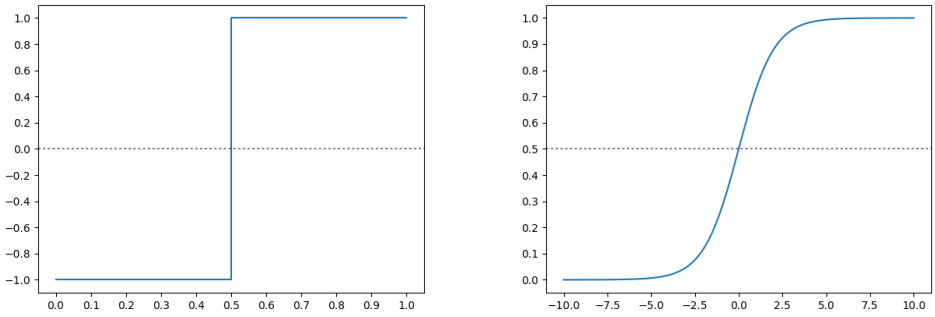
$$\sigma(z) = \frac{1}{1 + \exp(-\sum_{j=1}^{m} w_j {x_j}^{(i)} + b)} \tag{2.16}$$

The output of the sigmoid function (that is a real value between 0 and 1) is then compared with the threshold to decide whether the data sample belongs to the first class or the second one (the values of the target classes tend to be 1 and 0 in sigmoid neuron). The output of the sigmoid function can be interpreted as a conditional probability that a certain data sample (i) belongs to the positive class.

$$P(y^{(i)} = 1 | x^{(i)}) \tag{2.17}$$

If the shape of the sigmoid function is compared to the hard-limiter that can be found in the perceptron also known as step function (Figure 2.6), it can be seen that the smooth shape of the sigmoid function can help in making the impact of small changes in the weights and bias to be small on the output resulting with a more efficient learning process [15].

The sigmoid function is part of a family of functions that tend to be used in neurons. This family of functions called activation functions,



(a) Step Function (b) Sigmoid Function

Figure 2.6: Shapes of Functions

$$\sigma(.) : \Re \rightarrow \Re \tag{2.18}$$

In addition to the sigmoid function, other functions can be found in literature such as the tan hyperbolic function [14],

$$\sigma(z) = tanh(z) \tag{2.19}$$

These functions tend to have certain properties that distinguish them from others [15, 13]. In addition to being smooth in shape, the functions are differentiable and their derivatives can be simply written in terms of the function itself making it easier to compute the derivatives [15]. This property gains its importance because of the fact that the updates of the weights and the bias incorporates the derivative of the activation function. This appears in the update value of the weights δw. This value has to be based on how close the output of the activation function that is a real value to the target class of the sample that is a binary one. This creates

the need for a measure of this accuracy that is called the cost function C(). The cost function can take several forms each with its own advantage [15, 13, 14]. It can be in the form of Square Error,

$$C(W) = \frac{1}{2}(t^{(i)} - y^{(i)})^2 \tag{2.20}$$

or Logistic Cost,

$$C(W) = -t^{(i)} \log(y^{(i)}) - (1 - t^{(i)}) \log(1 - y^{(i)}) \tag{2.21}$$

where $y^{(i)}$ is the output of the activation function,

$$y^{(i)} = \sigma(X^{(i)}, W) \tag{2.22}$$

Based on the previous loss functions, the update rule of the weights would become,

$$W = W - \alpha \nabla_W C \tag{2.23}$$

where $\nabla_W C$ is the partial derivative of the cost function with respect to the weights. If the cost function used in the neuron is the logistic cost and the activation function used is the sigmoid function, then this value would be calculated as the following:

$$\nabla_W C = \frac{\partial}{\partial W}\Big(- t^{(i)} \log \big(\sigma(X^{(i)}, W)\big) - (1 - t^{(i)}) \log \big(\sigma(X^{(i)}, W)\big)\Big) \tag{2.24}$$

$$\nabla_W C = \Big(t^{(i)} \frac{1}{\sigma(X^{(i)}, W)} - (1 - t^{(i)}) \frac{1}{1 - \sigma(X^{(i)}, W)}\Big) \frac{\partial}{\partial W} \sigma(X^{(i)}, W) \tag{2.25}$$

$$\frac{\partial}{\partial W} \sigma(X^{(i)}, W) = \frac{\partial}{\partial W} \frac{1}{1 + \exp\left(-W^T X^{(i)}\right)} \tag{2.26}$$

$$\frac{\partial}{\partial W} \sigma(X^{(i)}, W) = \frac{\partial}{\partial W^T X} \Big(\frac{1}{1 + \exp\left(-W^T X^{(i)}\right)}\Big) \frac{\partial}{\partial W} (W^T X^{(i)}) \tag{2.27}$$

$$\frac{\partial}{\partial W} \sigma(X^{(i)}, W) = \Big(\frac{1}{1 + \exp\left(-W^T X^{(i)}\right)}\Big)\Big(1 - \frac{1}{1 + \exp\left(-W^T X^{(i)}\right)}\Big)(X^{(i)}) \tag{2.28}$$

$$\frac{\partial}{\partial W} \sigma(X^{(i)}, W) = \sigma(X^{(i)}, W)(1 - \sigma(X^{(i)}, W))(X^{(i)}) \tag{2.29}$$

$$\nabla_W C = \Big(t^{(i)} \frac{1}{\sigma(X^{(i)}, W)} - (1 - t^{(i)}) \frac{1}{1 - \sigma(X^{(i)}, W)}\Big)\Big(\sigma(X^{(i)}, W)(1 - \sigma(X^{(i)}, W))(X^{(i)})\Big) \tag{2.30}$$

$$\nabla_W C = \Big(t^{(i)} \frac{\cancel{\sigma(X^{(i)}, W)}\big(1 - \sigma(X^{(i)}, W)\big)X^{(i)}}{\cancel{\sigma(X^{(i)}, W)}} - (1 - t^{(i)}) \frac{\sigma(X^{(i)}, W)\cancel{\big(1 - \sigma(X^{(i)}, W)\big)}X^{(i)}}{\cancel{\big(1 - \sigma(X^{(i)}, W)\big)}}\Big) \tag{2.31}$$

$$\nabla_W C = \Big(t^{(i)}\big(1 - \sigma(X^{(i)}, W)\big)X^{(i)} - (1 - t^{(i)})\sigma(X^{(i)}, W)X^{(i)}\Big) \tag{2.32}$$

$$\nabla_W C = t^{(i)}X^{(i)} - \cancel{t^{(i)}\sigma(X^{(i)}, W)X^{(i)}} - \sigma(X^{(i)}, W)X^{(i)} + \cancel{t^{(i)}\sigma(X^{(i)}, W)X^{(i)}} \tag{2.33}$$

$$\nabla_W C = \big(t^{(i)} - \sigma(X^{(i)}, W)\big)X^{(i)} \tag{2.34}$$

$$W = W - \alpha\big(t^{(i)} - \sigma(X^{(i)}, W)\big)X^{(i)} \tag{2.35}$$

where α is the learning rate and takes a value between 0 and 1. This process is repeated until the value of the loss function is smaller than a certain threshold ϵ. This algorithm is called Gradient Descent [15].

Perceptrons and sigmoid neurons can in fact be very effective in classifying linearly separable data. However, when it encounters linearly non-separable ones they tend to behave poorly. One of the simplest linearly non-separable problems is the XOR problem that is represented in Figure 2.7. In this problem, it is clear that there cannot be any linear boundary that has the ability to separate the two classes. This created a need for more advanced algorithms that have the ability to create non-linear hyperplanes as the one that can be seen in Figure 2.8 to be able to solve such complex problems.

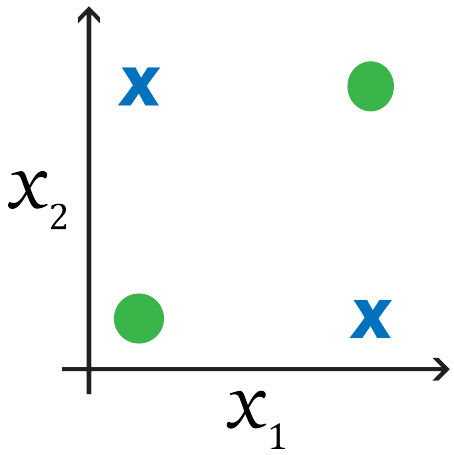


Figure 2.7: The XOR Problem

2.2.3 Multi-Layer Perceptrons

Researchers have found that connecting neurons in a network architecture can in fact solve non-linear problems. By taking the output of one neuron and feed it as an input to another one by creating consecutive layers of neurons, creates a hierarchical way of learning such that each layer adds on top of what was learned

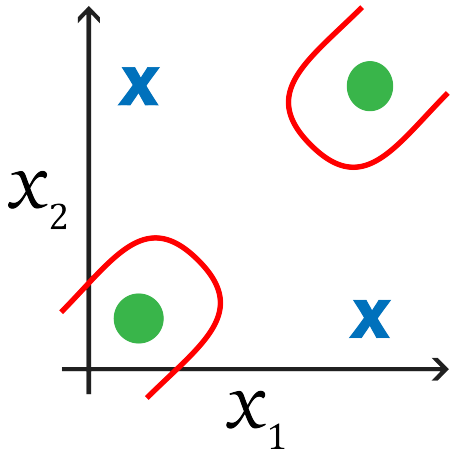


Figure 2.8: The XOR Non-Linear Solution

in the layer before [15]. This technique is called Multi-Layer Perceptrons (MLPs). Despite the fact that the name of technique contains perceptron, however, the type of neurons is not limited to them [13]. In literature, MLPs can be refereed to as Artificial Neural Networks (ANNs) to avoid any confusion with the word perceptron [2]. The structure of ANNs consists of three main components that can be seen in Figures 2.9 and 2.10: input layer, output layer and hidden layers. In the input layer, each neuron corresponds to an input variable and the arrows between that neuron and the neurons in the next layer represents the weight of that link. In hidden layers, each neuron represents a weighted sum of its inputs in addition to the bias followed by an activation function. The term hidden refers to the fact that this layer is neither an input layer nor an output one. There can be several consecutive hidden layers in the network. Furthermore, the number of neurons in each hidden layer can be different. These two hyperparameters can differentiate between networks' architecture and researchers tend to spend effort in order to find the best one that suits the problem they are trying to solve.

Of course, there is a trade off between the complexity of the network and the time it requires to train that researchers always try to balance based on their problem [14]. In the output layer, depending on the problem that the network is trying to solve, there can be more than one output neuron. If the problem is a classification problem between two classes, then one output neuron would be sufficient. However, some problems require the classification between multiple classes. In this case, each class would have its own neuron such that if the network classified the input sample to be of certain class, then the corresponding neuron would output 1 while the rest would output 0s in a one-hot encoding manner [15].

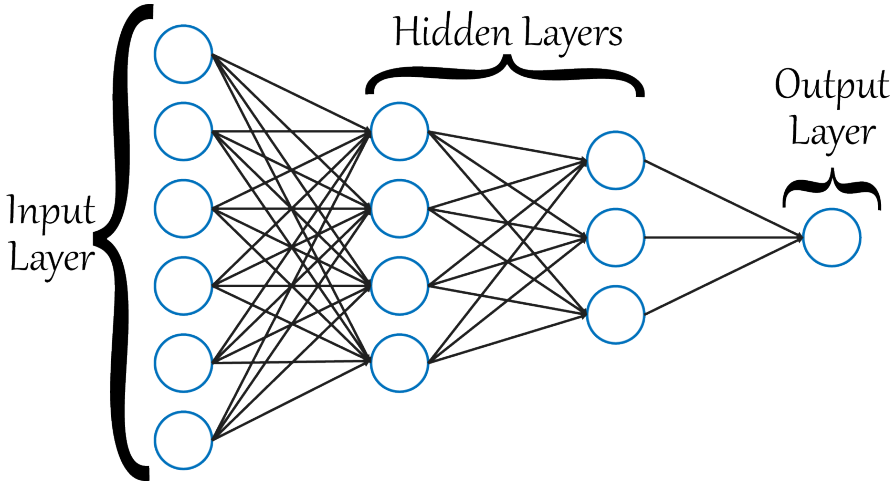


Figure 2.9: Multi-Layer Perceptron

Before discussing the way ANNs learn, it is essential to define a notation that can be followed throughout the explanation of the learning algorithm. The input to the network is a data sample represented as a vector of m variables denoted by $X^{(i)}$ where i is the sample number that ranges between $1, 2, \cdots, n$. An input variable will be denoted by ${x_j}^{(i)}$ where i is the sample number and j is the index

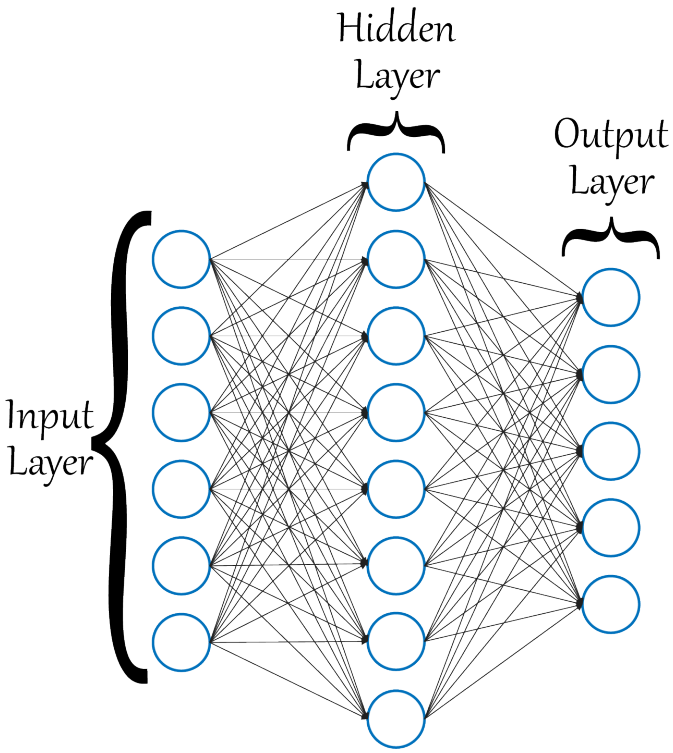


Figure 2.10: Multi-Layer Perceptron

of the variable that ranges between $1, 2, \cdots, m$. Each data sample is labeled with its true class that it belongs to. These labels are used as targets while training the ANN. The target of each data sample is denoted by $t^{(i)}$. All the targets are stored in a vector of n elements denoted as T. If the output of the network is one-hot encoded to classify multiple classes of size o, then in this case T will be a matrix of size $n \times o$ and $t^{(i)}$ will become a vector of o elements denoted by $t_j^{(i)}$. Since there will be different weights between neurons in each consecutive layers, let w_{jk}^l denote the weight for the connection from the k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer. Furthermore, the matrix W^l refers to all the weights in the l^{th} layer such that the weight w_{jk}^l is stored in the j^{th} row and k^{th} column. In these ANNs, it will be assumed that the biases are represented

separately from the weights. They will be referred to as b_j^l for the bias of the j^{th} neuron in the l^{th} layer. The set of biases in layer l will be referred to with the vector b^l. The symbol a_j^l will be used to denote the activation of the j^{th} neuron in the l^{th} layer. Those activations represent the outputs of neurons in both the hidden and output layers. The set of activations in layer l is denoted by the vector a^l. It is worth noting that the activations a^l are related to the activations of the previous layer $a^{(l-1)}$. However, before showing that, it is important to define the vectorized version of the activation function $\sigma()$. This version takes as input a vector and then applies the activation function on each element of the vector. This activation function will be denoted by $\sigma_e()$ as an element-wise application of the function. By returning back to the activations, the set of activations a^l can be defined as the following,

$$a^l = \sigma_e(W^l a^{(l-1)} + b^l) \tag{2.36}$$

Moreover, it is useful to define another quantity denoted by z^l that is referred to as the weighted inputs to the activations a^l such that,

$$z^l = W^l a^{(l-1)} + b^l \tag{2.37}$$

$$a^l = \sigma_e(z^l) \tag{2.38}$$

Each element of the vector z^l is denoted by z_j^l, which is just the weighted input

to the activation function for the j^{th} neuron in the l^{th} layer and can be defined as the following,

$$z_j^l = \sum_{k=1}^{p} w_{jk}^l a_k^{(l-1)} + b_j^l \tag{2.39}$$

,where p is the number of neurons in the $(l-1)^{th}$ layer.

One of the useful operations that will be used in the learning algorithm is the Hadamard product which is an element-wise product that can be applied on two vectors of the same size. The symbol that will be used to refer to this product is $\odot$ such that,

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \times 3 \\ 2 \times 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix} \tag{2.40}$$

The last thing to be defined is the cost function C(). In this learning algorithm the cost function will be Mean Square Error (MSE) that has the following formula,

$$C(W, b) = \frac{1}{2n} \sum_{i=1}^{n} \|t^{(i)} - a^L(X^{(i)})\|_2^2 \tag{2.41}$$

,where L is the number of layers, n is the total number of data samples and $a^L(X^{(i)})$ is the activations of the output layer when the input is $X^{(i)}$. This formula computes the average cost function across all data samples. In case the cost function was to be computed for one sample, it will be of the form,

$$C(W,b) = \frac{1}{2}\|t^{(i)} - a^L(X^{(i)})\|_2^2 = \frac{1}{2}\sum_{j=1}^{o}(t_j^{(i)} - a_j^L(X^{(i)}))^2 \tag{2.42}$$

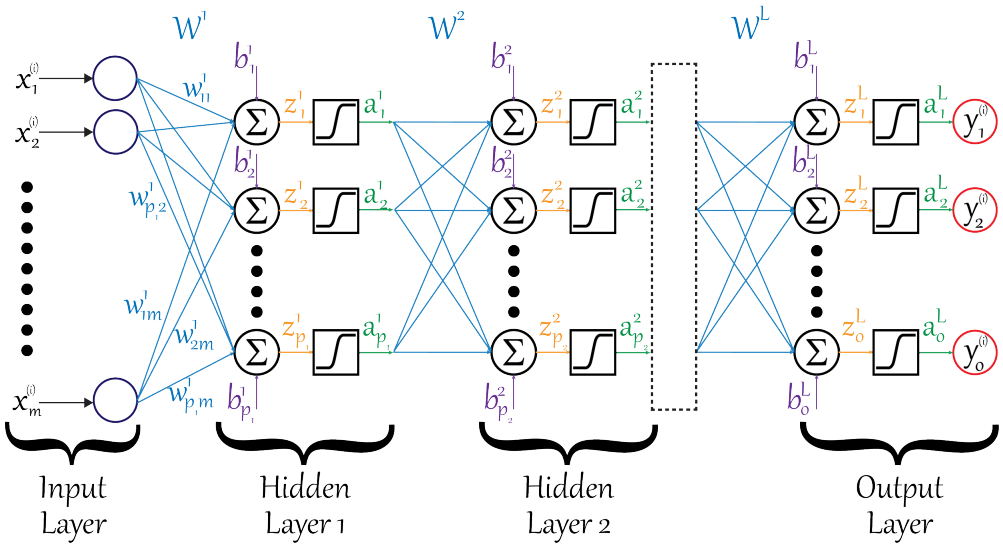


Figure 2.11: Multi-Layer Perceptron Notations

After all these notations and definitions, it is time to discuss how ANNs learn to classify data. In fact, they use a modified version of gradient descent algorithm. This modification allows the network to propagate the error in the classified data all the way back to the first layer such that all the weights and biases can update to adapt to better classification. The algorithm that was first introduced in 1974 is called Backpropagation [16]. However, it was not appreciated until 1986 when Rumelhart et al. used it in their paper to train different architectures of ANNs to solve several problems that were thought to be unsolvable [17].

Backpropagation algorithm tries to find how does the change in the network's weights and biases impact the cost function where the ultimate goal is to minimize it such that the difference between the real target values and the predicted ones is

minimal. The way this can be achieved is by calculating the partial derivative of the cost function with respect to each and every weight and bias in the network which are $\partial C/\partial w^l_{jk}$ and $\partial C/\partial b^l_j$ respectively. With these quantities, the network would be able to update its weights and biases using the following equations,

$$w^l_{jk} = w^l_{jk} - \alpha \frac{\partial C}{\partial w^l_{jk}} \tag{2.43}$$

$$b^l_j = b^l_j - \alpha \frac{\partial C}{\partial b^l_j} \tag{2.44}$$

However, to be able to calculate these quantities, backpropagation relies on the error in each neuron in the network. The error in neuron j in layer l which will be referred to as δ^l_j can be defined as the following,

$$\delta^l_j = \frac{\partial C}{\partial z^l_j} \tag{2.45}$$

and δ^l is the vector of errors in the l^{th} layer. With that said, starting from the output layer, the error in this layer δ^L_j can be defined and computed using the chain rule as the following,

$$\delta^L_j = \frac{\partial C}{\partial z^L_j} = \frac{\partial C}{\partial a^L_j}\frac{\partial a^L_j}{\partial z^L_j} = \frac{\partial C}{\partial a^L_j}\sigma'(z^L_j) \tag{2.46}$$

The two parts that form the error can indicate two different ratings. The first term on the left gives an indication on how fast the cost function changes with respect to the changes in the j^{th} neuron. On the other side, the second term

measures the effect of the changes in the weighted sum on the activation function $\sigma()$ in the j^{th} neuron. These two terms can be easily calculated. The term on the left is simply equal to the following,

$$\frac{\partial C}{\partial a_j^L} = a_j^L - t_j^{(i)} \tag{2.47}$$

when the cost function is equal to equation (2.42).

The second term depends on the activation function used. If the sigmoid function was used, then its derivative can be calculated using the set of equations (2.26 - 2.29). However, for simplicity it will be referred to as $\sigma'(z_j^L)$ within the upcoming equations. To further simplify the equations, they can be written in matrix notation instead of the component notation one that has been used. Therefore, the vector of errors δ^L would be defined as,

$$\delta^L = \nabla_{a^L} C \odot \sigma'_e(z^L) = (a^L - t^{(i)}) \odot \sigma'_e(z^L) \tag{2.48}$$

The next essential part in backpropagation is to define the errors δ^l in layer l in terms of the errors in the next layer $\delta^{(l+1)}$. This can be computed using,

$$\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot \sigma'_e(z^l) \tag{2.49}$$

where $(W^{l+1})^T$ is the transpose of the weight matrix in the $(l+1)^{th}$ layer. The use of the transpose can be thought of as if the errors in the $(l+1)^{th}$ layer is backpropagated to the output of the neurons' activations in the $(l)^{th}$ layer and

then using the Hadamard product, the errors are further propagated through the activation function of that layer. It is worth noting that equations (2.48) and (2.49) are sufficient to calculate the errors δ^l of any layer l all the way back to the input layer. This can be achieved by starting from the output layer and applying equation (2.48) and then repeatedly applying equation (2.49) until reaching the desired layer.

Since the errors δ^l in any layer can now be calculated, it is time to calculate $\partial C/\partial w^l_{jk}$ and $\partial C/\partial b^l_j$. Starting with the first one and using the chain rule it can be defined as the following,

$$\frac{\partial C}{\partial w^l_{jk}} = \frac{\partial C}{\partial z^l_j}\frac{\partial z^l_j}{\partial w^l_{jk}} \tag{2.50}$$

From equations (2.39) and (2.45), this will be equal to,

$$\frac{\partial C}{\partial w^l_{jk}} = \delta^l_j a^{(l-1)}_k \tag{2.51}$$

Similarly, it can be found that,

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{2.52}$$

Using the last two equations, weights and biases updates equations (2.43) and (2.44) can be rewritten to be,

$$w^l_{jk} = w^l_{jk} - \alpha\delta^l_j a^{(l-1)}_k \tag{2.53}$$

$$b_j^l = b_j^l - \alpha \delta_j^l \tag{2.54}$$

This concludes the backpropagation algorithm which is considered until today to be the workhorse of learning algorithms in ANNs. By depending mainly on four essential equations [(2.48),(2.49),(2.53),(2.54)], ANNs can be able to learn not only to classify data but even to predict new ones which is known as regression. There have been several studies in the literature showing the success in using ANNs in solving a variety of problems including medical and heart diagnoses [18, 19], handwriting recognition [20, 21], speech recognition [22], face recognition [23, 24] and damage detection of bridge structures [25].

However, despite its effectiveness and success, ANNs have its own share of issues and drawbacks. One of the well known issues related to ANNs is what is called the gradient vanishing problem [15]. Its ramification can be seen in Figure 2.12 that shows the rate of change of the gradient while backpropagating the error through the hidden layers. It can be noticed that the neurons in the second layer which is closer to the output layer has a faster rate of change. This means that these neurons can in fact learn faster than the ones in the first layer. This behaviour gets worse as the number of hidden layers increases. In Figure 2.13, it can be noted that the difference between rate of change in the last hidden layer and first one decreased dramatically by just adding two extra hidden layers when compared to Figure 2.12 to reach around 100 times slower in the first layer. This issue occurs because as the error being backpropagated through the layers, it gets multiplied

by other terms in each layer. Those terms and error tend to have values less than 1, which makes the error become infinitesimal such that the first layers would not be able to learn much.

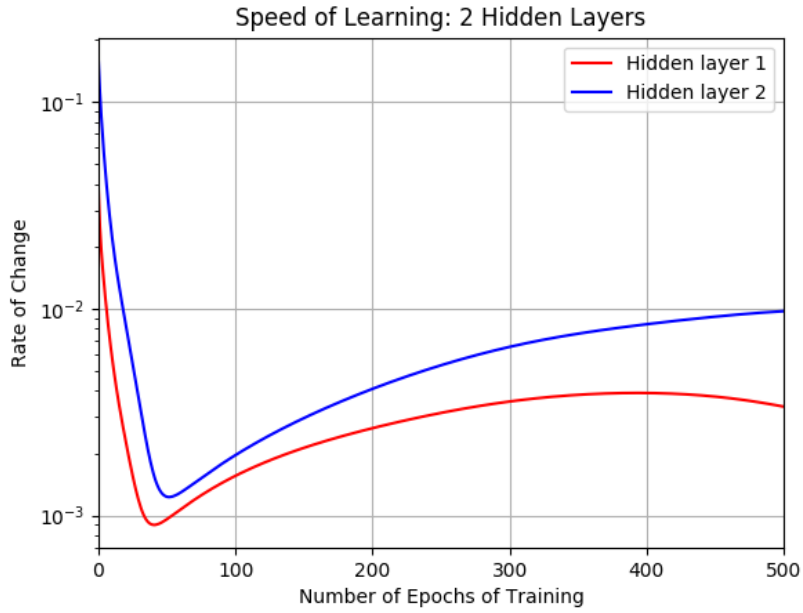


Figure 2.12: Training Speed in ANN With Two Hidden Layers

One of the drawbacks of ANNs is related to its structure and how the inputs are fed to the network specially when they are images [15]. Although ANNs have shown noticeable advancements in problems related to images like recognition and classification. However, feeding an image to an ANN causes a loss in spatial information which limits and prevents the network from reaching the optimal solution. To put things in perspective, Figure 2.14 shows a simplified example of a network that is fed with face images for classification. Because of the way the input neurons are stacked in ANNs, the input image therefore has to be preprocessed before feeding it to the network. What the preprocessing step does

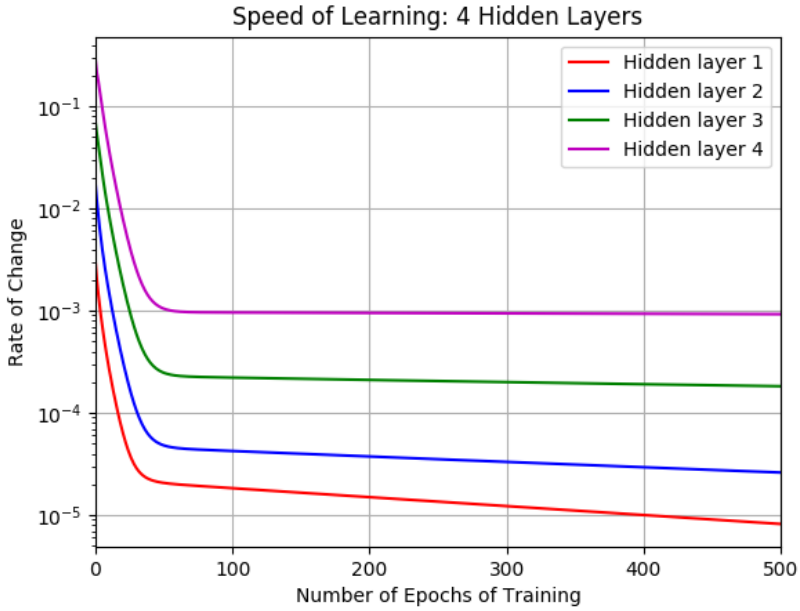


Figure 2.13: Training Speed in ANN With Four Hidden Layers

is that it flatten the image by stacking the pixels of the image in a vector form (Figure 2.14 does not show that due to limitation in representation). When this happens, spatial information will be lost because pixels that were near each other are no longer are and therefore the structure of the image has totally vanished. Due to that, researchers start to think about how to change ANNs such that they can preserve these spatial information.

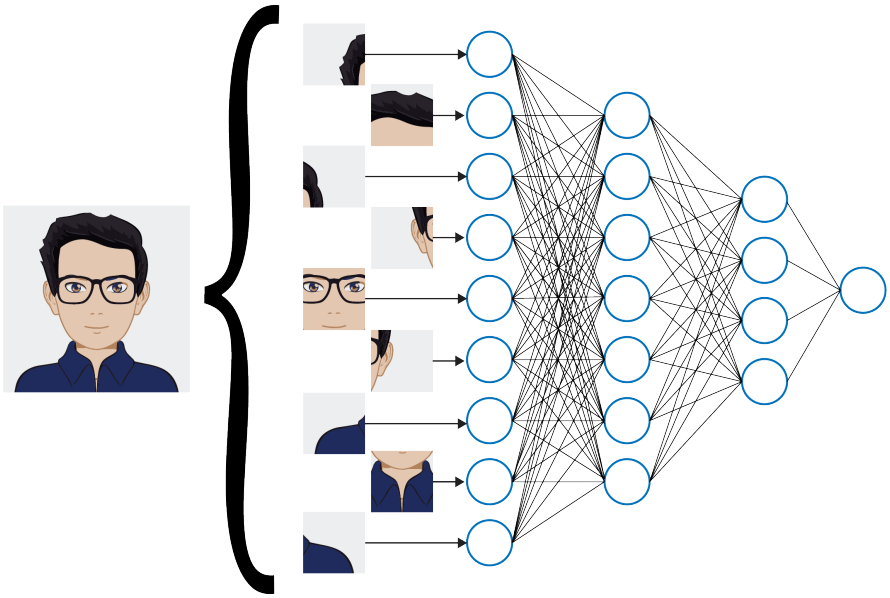


Figure 2.14: ANNs Architecture Drawback

2.2.4 Deep Convolutional Neural Networks

One of the well-known algorithms in DL that handles images and extracts meaningful features is the Convolutional Neural Networks (CNNs). These networks are biologically inspired variants of MLPs. The main inspiration of CNN connectivity pattern within its neurons is the shape animal visual cortex are organized. CNNs can be explained with three main ideas: local receptive fields, shared weights and pooling [26].

Local receptive fields

In fully-connected layers that appear in MLPs, the inputs are represented as a straight line of neurons. However, in a CNN, the input is considered as a square of neurons. The values of those neurons correspond to the pixel intensities of the image that is used as an input. The architecture is constructed as usual by con-

necting the input neurons to a layer of hidden neurons. However, the connection will not be established between every input pixel to every hidden neuron creating a fully-connected layer. Instead, the connections will be established in localized regions of the input image that are small in size. This will create a connection between each neuron located in the first hidden layer and small set of input neurons that create a region. The size of the region varies in literature depending on the application. Assuming that the size is 4x4. Therefore, this corresponds to 16 input pixels (Figure 2.15). The corresponding window region in the image used as an input is called the local receptive field for the hidden neuron. Once the connection is established it is then assigned a weight. The main purpose of the neurons in the first hidden layer is to learn to analyze its corresponding local receptive field. By sliding the local receptive field one pixel at a time, the first hidden layer can be built. However, the local receptive field can be moved by more than one pixel. In fact, in literature, the number of skipped pixels is called the stride length and is considered one of the parameters that researchers experiment with[27][28].

Shared weights and biases (Parameter Sharing)

An important difference between CNNs and MLPs is that the weights connected to the hidden neurons and the bias are the same across all the neurons. This implies that a certain exact feature will be detected using all the neurons in the hidden layer however each one in different location in the input image. A feature can be in the form of certain pattern that would appear in the input image for

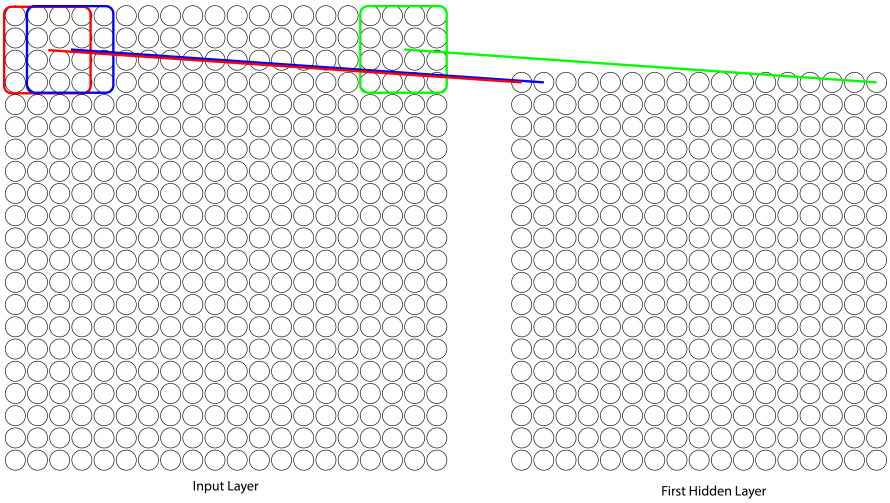


Figure 2.15: CNN Local Receptive Fields

example a vertical edge. This mapping between the layer of input neurons and the hidden layer is referred to as a feature map. Furthermore, the weights and the bias that define the feature map are called the shared weights and the shared bias respectively. In literature, sometimes they are referred to by shared parameters and those parameters form what is called a filter [15]. A network with a structure of one hidden layer with single filter can be used just to detect a single localized feature. Therefore, in order to apply image recognition, a set of feature maps must be utilized. Based on that, a complete convolutional layer consists of several feature maps that are different[27][28].

Pooling layers

Pooling layers are another set of layers that are contained in CNNs in addition to the convolutional layers. They tend to be used immediately after convolutional layers. Simplifying the information derived from the output of the convolutional

layer is considered to be the purpose of pooling layers. This can lead to reduce the amount of parameters needed and the computation in the network in addition to controlling over-fitting. Therefore, a pooling layer takes the output of each feature map from the convolutional layer and prepares a smaller reduced feature map such that each unit in the pooling layer may summarize a set of neurons in the previous layer. Max-pooling is considered one of the most common procedures used for pooling. A pooling unit in a max-pooling layer would simply output the maximum activation in a small input region as shown in Figure 2.16. If the architecture of previous convolutional layer consists of several feature maps, therefore, there will be a max-pooling layer for each feature map separately. In literature, max-pooling is one among other techniques used for pooling. An alternative common approach that is used in pooling is known as L_2-pooling, where in this technique instead of considering the maximum activation of a small region of neurons, the square root of the sum of the squares is taken [15].

However, despite all the advantages mentioned earlier, yet it has another advantage that can impact DQN negatively which is translation invariance. This feature gives the network the ability to detect the same shape regardless of its position in the image. For example, if the network is trained to detect faces, it does not matter if the face is in the center or shifted off to one of the edges. In both cases the network will be able to find it. However, in DQN the position is very important such that it can have a huge impact on the performance of the agent. Consider the Pong game, the position of the ball highly influences the next actions to be

taken by the agent. If the ball is in the upper half while the paddle of the agent in the bottom, then the agent has to move up to be able to hit it and vice versa. Therefore, the position can change the expected outcome and that is the reason for omitting the pooling layer in DRL algorithms [27][28].

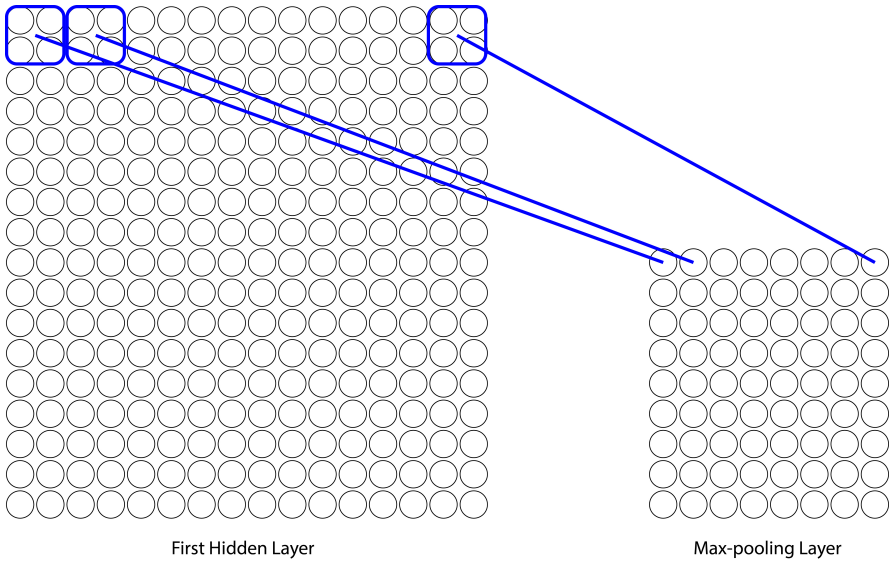


Figure 2.16: CNN Max-Pooling Layer

Convolutional Neural Network Architecture

A full CNN can be built using consecutive convolutional layers followed by pooling layers. However, the final layer would be a fully-connected layer where each neuron in the layer corresponds to certain predicted output. In literature, the architecture of CNN can vary based on the values of the hyper-parameters that shape the network[27][28]. An example of a CNN would look similar to Figure 2.17 which was applied to a screen-shot from Pong game environment.

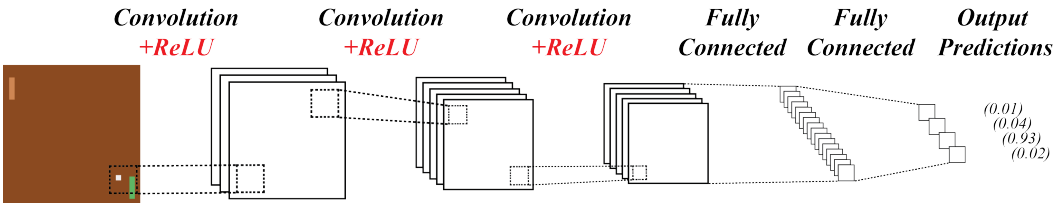


Figure 2.17: Convolutional Neural Network Architecture

2.2.5 Optimization Techniques

ADAM: A method for stochastic optimization

ADAM which is derived from adaptive moment estimation is a first-order gradient-based technique that was designed for the optimization of stochastic objective functions, in particular, those with high-dimensional parameters spaces [29]. Evaluating random mini-batches of data points along with inheriting function noise coming from data subsampling or other sources like dropout regularization can arise stochasticity in objective functions which ADAM was designed to handle. By utilizing the estimates of first and second moments of the gradients, ADAM computes individual adaptive learning rates for function parameters with little memory requirement. The advantages of ADAM are not limited to minimal memory fingerprint, it further includes a carefully chosen step size that is bounded approximately by the step size hyper-parameter where the algorithm performs naturally a form of step sizes annealing. This advantage is considered one of the important properties of ADAM's update rule. Furthermore, the utilization of initialization bias correction terms in ADAM helps in step size regularization since not correcting the bias can lead to very large step sizes and more often leads to divergence.

Moreover, the advantages of ADAM include that re-scaling the gradient does not affect the magnitude of parameter updates, stationary objectives are not required and it can handle sparse gradients. In fact, the latter two advantages of ADAM were inherited from the two popular optimization techniques: RMSProp and AdaGrad respectively. RMSProp with momentum is a variation of the original RMSProp that is closely related to ADAM. However, they vary in that RMSProp with momentum utilizes a momentum on the re-scaled gradient to generate the updates of its parameters whereas ADAM uses a running average of the first and second moments of the gradient to estimate the updates. Furthermore, RMSProp lacks any terms for bias-correction in opposite to ADAM. Nevertheless, the advantage of RMSProp comes in the ability to work with non-stationary objectives and on-line settings [30]. On the other hand, one of the well-suited algorithms to handle sparse gradients is AdaGrad [31]. In fact, AdaGrad can be considered a special case of ADAM where specific values are giving to the hyper-parameters β_1 and β_2 which control the exponential decay rate of the running averages that estimates the first and second moments respectively. β_1 is given a value of zero where β_2 is given a very close value to one such that (1- β_2) is infinitesimal and the learning rate α is replaced by an annealed version.

Algorithm 4: ADAM Pseudo-Code

Initialize θ_0 ;

Initialize 1^{st} moment vector, $m_0 \leftarrow 0$;

Initialize 2^{nd} moment vector, $v_0 \leftarrow 0$;

Initialize timestep $t \leftarrow 0$;

while θ_t *not converged* **do**

$t \leftarrow t + 1$;

$g_t \leftarrow \nabla f_t(\theta_{t-1})$;

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$;

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$;

$\hat{m}_t \leftarrow m_t / (1 - {\beta_1}^t)$;

$\hat{v}_t \leftarrow v_t / (1 - {\beta_2}^t)$;

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$

end

return θ_t

2.3 Testing Benchmarks

2.3.1 The Arcade Learning Environment

Arcade Learning Environment (ALE) was introduced by [32]. The main purpose of ALE was to be considered as an evaluation platform and environment to aid the development of general domain-independent AI technologies in addition to being a challenging problem. ALE provides an access interface to huge number of Atari 2600 game environments where all games are different and interesting in addition to being designed in away such that they can be a challenge for human players. ALE offers significant research challenges for a variety of AI learning techniques from RL to model learning and model based planning in addition to imitation learning, transfer learning, and intrinsic motivation. However, the most important contribution of ALE is that it provides a rigorous test-bed as a benchmark for evaluating and comparing approaches in solving those problems.

Pong Game

The pong game is a two-player competitive game that simulates table tennis sport in a two-dimensional perspective. Each player controls an in-game paddle by moving it vertically up and down trying to hit the ball to prevent it from passing him. Each time a player misses the ball, he loses and the opponent scores one point. The players compete on who to reach 21 points first (Figure 2.18). The implementation of Pong game in ALE gives the control of one of the players (the left side) to a hand-written basic AI while the other player is left to be

controlled by researchers' developed agents.

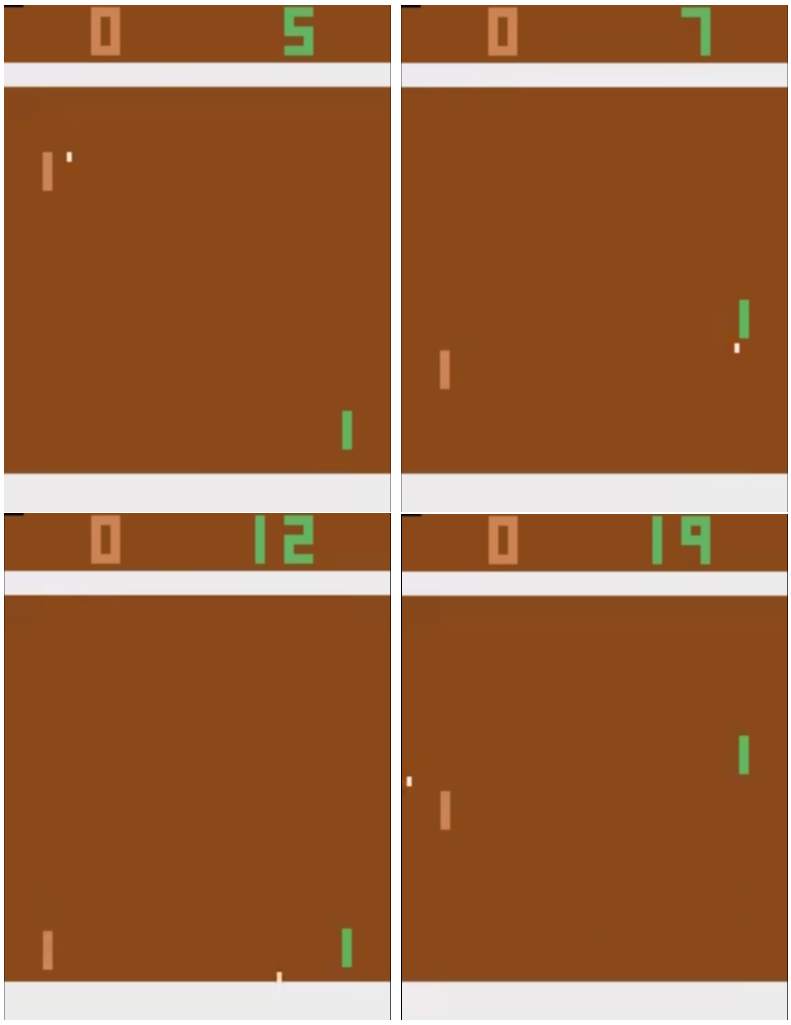


Figure 2.18: Pong Game

2.3.2 ViZDoom

Atari 2600 games that were introduced in ALE were considered great tools to test and benchmark a variety of AI algorithms. However, they suffer from the fact of being simple to reflect real-world objectives. This is because their graphics are non-realistic whether they are 2D environments or 3D ones with third-person perspective. In [33] and [34], they proposed a novel benchmarking platform for RL research that uses raw visual information that is semi-realistic 3D world seen from first-person perspective. Because it was inspired and built based on the well-known classic game Doom, the tool was named ViZDoom. Using frames, researchers were able to develop intelligent bots that can play the game. ViZDoom provides several types of frames extracted from the game environment in real time. The frames can be simply the screen buffer that appear on the screen to the players or they can contain more information as how far the objects are from the player presented in what they called a depth buffer. Both types can be seen in Figure 2.19. ViZDoom is considered fast in terms of performance and highly customizable via a convenient mechanism of user scenarios. The environment provides several ready-made scenarios for learning bots in addition to giving the developers and researchers the ability to define customizable ones. Among predefined scenarios, two of them were used in the experiments of this thesis. The first scenario is a basic one that is based on move-and-shoot task. It takes place in a rectangular room starting from the center where the agent is spawned standing opposite to an enemy monster that neither moves nor attacks.

The objective is to kill this monster that is randomly spawned as fast as possible to get the highest score (Figure 2.20). The player can take one of three actions or a combination of them: moving to the right, moving to the left or attacking. The second scenario takes place in a round hall where the player is spawned in the center and multiple enemies are spawned around him at the edge of the hall. The objective is to kill those enemies before they reach the center and start attacking the player. Each time the player been hit, the health meter will decrease until it reaches zero. The available actions for the player in this scenario are a little bit different. In contrast to the first scenario, the player can no longer move left and right and therefore can not leave the center, However, it can turn left and right to be able to attack the enemies coming from around the hall. (Figure 2.21).

Figure 2.19: ViZDoom Screen Buffer (left) and Depth Buffer (right)

Figure 2.20: ViZDoom First Scenario: Basic

Figure 2.21: ViZDoom Second Scenario: Defend The Center

2.4 Literature Review

TD-Gammon is one of the early works that utilized RL and ANNs [35]. TD-Gammon's approach was by adopting temporal-difference RL to play backgammon game which is considered to be a zero-sum game where two players will play and eventually one of them will be a winner while the other will be a loser. The system was trained from random initialization using temporal-difference RL. This was achieved through self-play. TD-Gammon reached a master-level performance after playing 1.5 million games in training. This allowed it to greatly surpassing all previous programs. However, TD-Gammon relied on ANNs with raw board input as well as hand-designed features for board representation. This gave it the ability to remove any search requirement and provided it with the ability to simply pick the move that would result in a position with the best evaluation score from any given position. However, despite this achievement, yet it was dependent on the human intervention with the hand-crafted rules. Nevertheless, this work was considered to be the motivation behind more sophisticated results that came afterwards since it was the first to apply RL for solving a complex game as backgammon compared to other researches at that time.

The development of DL area in ML has gained huge popularity due to its enormous improvements in the field of recognition and classification. This was due to the ability of DL algorithms in dealing with high-dimensional sensory inputs and extracting meaningful features. This performance of DL motivated DeepMind from Google in 2013 to introduce what they claimed to be the first RL

model that successfully utilizes DL as non-linear value function approximator to autonomously learn control policies [3]. At that time, most successful RL models depended on hand-crafted features. Furthermore, the value functions used with these models were restricted to linear value functions with some exceptions. One of the earliest trials in introducing ANNs in RL was Neural Fitted Q-Learning [36], [37] & [38]. However, the reason for DL not to gain popularity as function approximators comes from the fact that DL algorithms assume independency of the data samples. However, this is not the case in RL where sequences of highly correlated states can be encountered more often. Furthermore, DL algorithms assume a fixed distribution whereas in RL the distribution of the data changes rapidly as new behaviors are learned. Therefore, in order to solve the problems of data correlation and non-stationary distributions DeepMind had to use what is called experience reply mechanism. This mechanism simply samples random batches uniformly from stored previous transitions and use them instead of current consecutive ones. This was very effective in smoothing the training distribution over various past behaviors and it was very important for making the algorithm run successfully. In fact, when learning directly from consecutive transitions, undesirable feedback loops may occur and therefore the parameters of the network may get stuck in what can be considered a poor local minimum or even diverge. The use of experience reply can help averaging the behavior distribution over several previous states and that can lead to a smooth learning and help in avoiding oscillations or even divergence in the network's parameters. The use of experience

reply forces the model to learn off-policy and this motivated the use of Q-learning in the first place. This comes from the fact that learning on-policy means that parameters are updated using a data sample dependent on the current values of the parameters. However, this is not the case when it comes to experience reply since the data sample is randomly chosen from previous ones and therefore it was generated using different parameters from the current ones. The size of the memory buffer used in this mechanism was chosen to be one million. This limited size along with the uniform sampling have some drawbacks. The way of selecting transitions even though it is random yet it is considered to be naive since it does not differentiate between important transitions and the limitation in size forces the algorithm to overwrite old transitions with new ones despite their importance or rareness. This limitation was later addressed in another research proposed by [39] who replaced the random sampling with a prioritized experience reply buffer that would rank the experiences based on the rewards achieved in them and how rare they are and when the algorithm needs to store new ones it would get rid of the least important ones. The DQN algorithm is considered a model-free and off-policy that was built to learn control policies from RL environments by learning directly from high-dimensional sensory input using a CNN trained with Q-learning algorithm. Since the model was built using a variation of Q-Learning that uses a deep CNN it was called Deep Q-Network (DQN). The input to the CNN was a preprocessed image represented as raw pixels extracted from Atari games screen and the output was Q-value estimations of all possible actions. The

advantage of this architecture in opposite to having the action as part of the input and the output is one single Q-value that corresponds to the input action is that it requires only one single forward pass through the network to compute the Q-values for actions in any given state compared to the other architecture that would require a separate forward pass through the network for each action, and this cost increases linearly with the number of actions. By referring back to the raw pixels of images taken from game's screen and fed to the agent as inputs, it can be realized that limiting the input to current screen can overcomplicate the task since it will be partially observable especially when knowing that many states in the emulator can be considered perceptually aliased which means understanding the current situation fully is impossible using only the current screen. In order to overcome this problem, the state of the MDP is considered to be a sequence of actions and observations and therefore the algorithm is supposed to learn strategies for playing the game based on those sequences. This can be reflected in the CNN by feeding it with a stack of consecutive frames instead of one single frame. In fact, the number of frames used was four frames. The network was trained to approximate Q-values by minimizing a sequence of loss functions. This loss was computed using the difference between the estimated Q-values and target ones. However, those target values in contrast to the ones used in supervised learning they change periodically. In fact, those target Q-values are generated using a second Q-network that the algorithm updates once in a while to match the values in the primary network.

$$(y_t - Q(s_t, a_t; \theta))^2 \tag{2.55}$$

$$y_t{}^{DQN} = r_t + \gamma \text{max}_{a'} \hat{Q}(s_{t+1}, a'; \theta^-) \equiv r_t + \gamma \hat{Q}(s_{t+1}, \text{argmax}_a \hat{Q}(s_{t+1}, a; \theta^-); \theta^-) \tag{2.56}$$

The reason for fixing the parameters of the target Q-network for certain amount of time steps (DQN algorithm used 100k) is to insure the stability of the primary network. Otherwise, if the values of both Q-networks update every time step, it can lead to falling into feedback loops between the estimated and target Q-values. The optimization of the loss function was achieved using stochastic gradient descent. In particular, the algorithm was equipped with RMSProp optimization technique for updating the weights. The developed algorithm was applied to seven Atari 2600 games from the ALE and then later it was applied to all the games in ALE [40]. The algorithm was able to learn how to play those games without any adjustment made to the architecture or tuning of the learning algorithm. The results of the algorithm outperformed all preceding methods on most games and surpassed a human expert on some of them. This algorithm is currently considered to be the baseline for all the research conducted in the area of DRL since it provides a new prospective to the problem and still has an area for improvements.

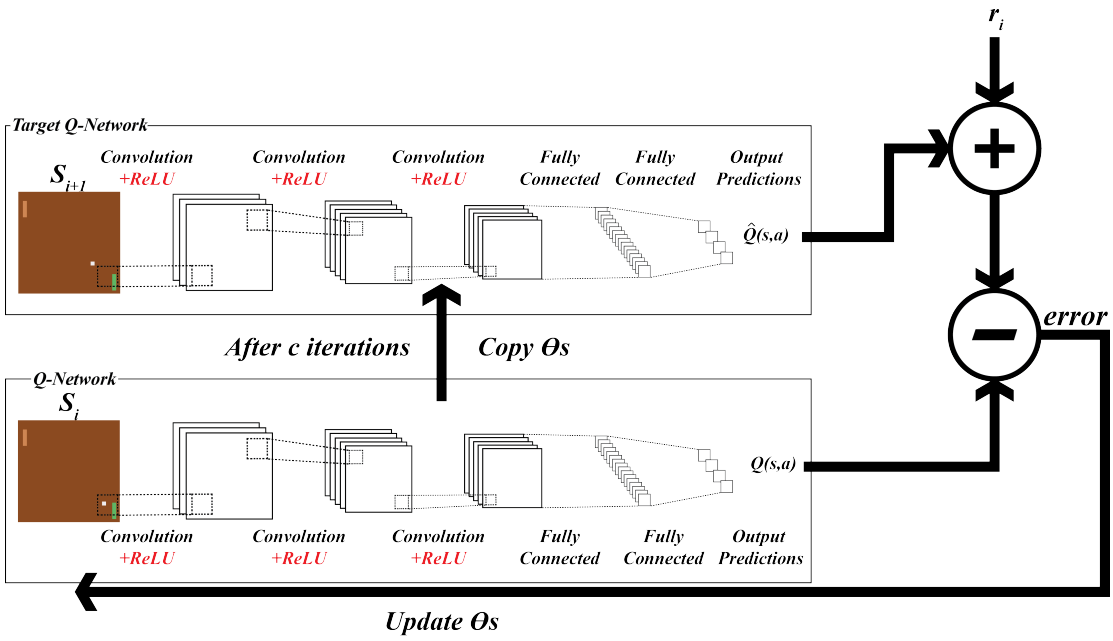


Figure 2.22: Q and Target Q-Networks

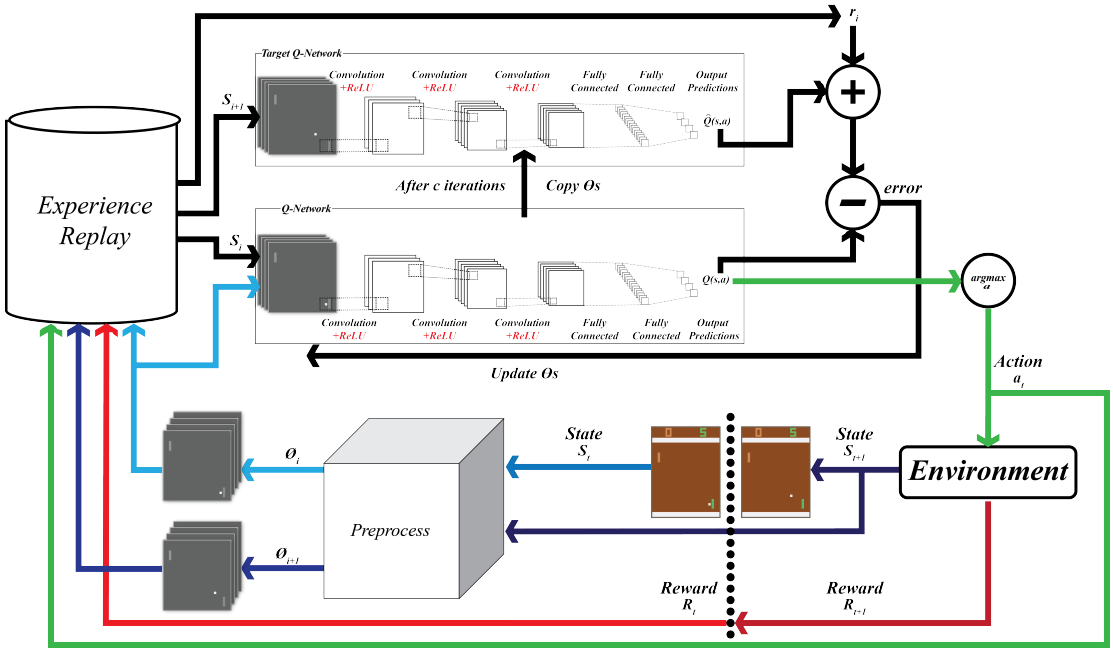


Figure 2.23: Deep Q-Network

Algorithm 5: DQN Pseudo-Code

Initialize replay memory $\mathcal{D}$ to capacity $\mathcal{N}$;

Initialize action-value function Q with random weights θ ;

Initialize target action-value function $\hat{Q}$ with random weights $\theta^{-} = \theta$;

for *episode* $\leftarrow 1$ **to** $\mathcal{M}$ **do**

Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$;

for $t \leftarrow 1$ **to** $\mathcal{T}$ **do**

With probability ϵ select a random action a_t otherwise select

$a_t = argmax_a Q(\phi(s_t), a; \theta)$;

Execute action a_t in emulator & observe reward r_t and image x_{t+1} ;

Set $s_{t+1} = s_t, a_t, x_{t+1}$ & preprocess $\phi_{t+1} = \phi(s_{t+1})$;

Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$;

Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ in $\mathcal{D}$;

Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{for non-terminal } \phi_j + 1 \end{cases}$

Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect

to the network parameters θ ;

Every $\mathcal{C}$ steps reset $\hat{Q} = Q$;

end

end

Despite the fact that Q-learning is considered one of the most well-known algorithms in RL along with its advantages. However, it suffers from an unrealistic behavior where it overestimates action values which has proven to lead to learning sub-optimal policies [41]. Several studies indicated that the source of these overestimations can be due to the insufficiently flexible function approximation [41], noise in the environment [42] or when the action values are inaccurate [4]. In fact, this inaccuracy can be practically found in any method during learning phase since the true values of the parameters to be learned are initially unknown regardless of whether the inaccuracy is due to estimation errors coming from non-stationarity, environmental error, function approximation or any other source which as a result can produce an upward bias. Furthermore, Q-learning has shown a tendency in preferring overestimated values over underestimated ones because of the maximization step over action values included in its update rule. DQN algorithm that depends on Q-learning has in fact inherited the same behavior even though it showed good results in learning Atari games. However, [4] has proven that when the estimations of the action values are close to the true ones or even underestimated, the model can achieve better results or even surpass human experts. The first to introduce a solution to the overestimation problem of Q-learning was [42]. The proposed algorithm was called Double Q-learning which was a variation from the original Q-leaning algorithm. The idea behind the proposed algorithm was based on the fact that the usage of the exact same values for both selecting and evaluating an action in the Q-learning max operator is causing this behavior

of overoptimistic value estimation. Decoupling the selection from the evaluation can in fact prevent the overestimations. In order to achieve this, the algorithm introduced a second Q function where in opposite to the original Q-learning, each Q-function is updated using the values of the other function. By following this method, it can be thought of as if the updated function is used to decide the greedy policy while the other function used to determine the value of the action or in other words evaluate the value of the policy fairly. The update of the two Q-functions was conducted randomly by assigning each experience to one of the functions to update its values and therefore, each function even though updated using the same problem however each one used different set of experience samples. This in fact can be considered an unbiased estimate of the value of the actions. The difference between Double Q-learning and the adapted version applied to DQN is that the original Double Q-learning was applied in a tabular setting whereas in Double DQN (DDQN) it was adapted to work with non-linear function approximators more precisely deep CNNs. Furthermore, in Double Q-learning two different Q-functions were used to update one of them using the other randomly using experiences encountered. On the other hand, DDQN used the same target network that is used in the computation of the loss function as a second value function even though it is not fully decoupled. However, it helped in reducing the number of networks required to apply Double Q-learning. The update of the target network remained the same as in DQN to be periodically copying the parameters of the primary network after certain number of time steps. The impact

of Double Q-learning when applied to DQN was not limited to better and more accurate action value estimation. In fact, it was extended to include better quality policies. The experiments have shown that results of DDQN in Atari games have exceeded the ones of DQN and got closer to scores of human players or even surpassing them in some cases. This have showed that DDQN is more stable in learning and more robust to the challenges presented by the environment which yielded better more general policies. On the other hand, the instabilities in DQN was in fact due to the overoptimism of Q-learning not due to the usage of off-policy learning with function approximation [4].

$$y_t^{\text{DoubleDQN}} = r_t + \gamma \hat{Q}(s_{t+1}, \text{argmax}_a Q(s_{t+1}, a; \theta); \theta^-) \tag{2.57}$$

Another suggested improvement over DQN was Dueling DQN that proposed by [43]. Their improvement was in terms of a new CNN architecture that leverages two streams such that the new architecture would be able to estimate two different functions: an action advantage function that is state-independent in addition to a state value function. These two estimated values are then combined together in an aggregating layer to produce the Q-values that can be used later in the RL algorithm. The intuition behind this new architecture is to give the network the ability to learn how valuable each state is without depending on the value of each action in that state and this can generalize the learning across all the actions and this can be beneficial in states where all its actions do not impact the environment in a relevant way. The change in the network architecture occur

es in the upper layers. The lower layers of Dueling DQN remained convolutional layers as the ones used in DQN. However, these layers are followed by two streams of two fully connected layers one for value function and one for action advantage function instead of one in DQN. The second hidden fully connected layer in the value stream has one output where the action advantage stream has a number of outputs the same as the number of actions. Then these two streams outputs are combined together to produce the Q-value estimations. This new architecture has shown an improvement in performance over DQN. Furthermore, since the input and output of the new architecture remained the same as the one used in DQN, it gave this architecture the advantage of using any improvements applied to DQN such as DDQN and others. Even though this solution managed in improving the performance of DQN. However, it increased the complexity of the model to some extent.

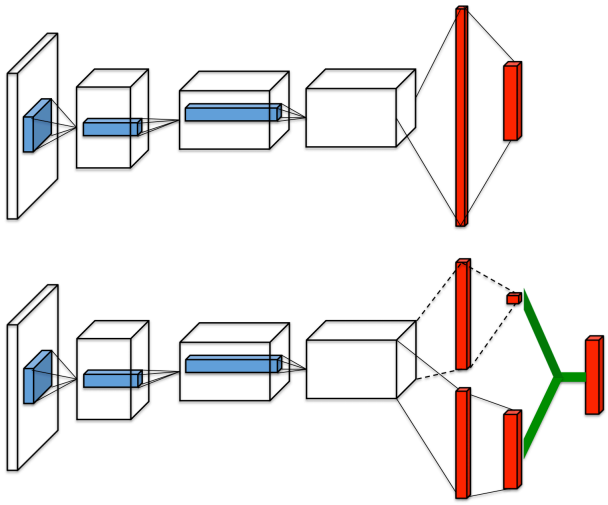


Figure 2.24: DQN (top) vs Dueling DQN (bottom) Network Architecture

One of the drawbacks of DRL using DQN was the limited memory and missing the ability to keep tracking of past experiences. To address this shortcoming, [44] proposed a solution where the first post-convolutional fully-connected layer in DQN was replaced with a recurrent Long Short-Term Memory (LSTM) one which added a recurrence to the model. The resulting Deep Recurrent Q-Network (DRQN), despite the fact that it is only fed with a single frame at each time-step, it can successfully integrate information through time and replicate DQN's performance on standard Atari games and partially observed games. Additionally, the performance of the proposed DRQN has shown to be a function of observability. This was observed when the model was trained with partial observations and then evaluated with incrementally more complete observations. However, in the opposite hand, when the model was trained with full observations and then evaluated with partial observations, the performance of DRQN showed a degradation to become less than the performance of DQN. Thus, given the same length of history, recurrent nets provide a viable alternative to using a stack of history frames in the DQN's input layer and can better adapt at evaluation time if the quality of observations changes.

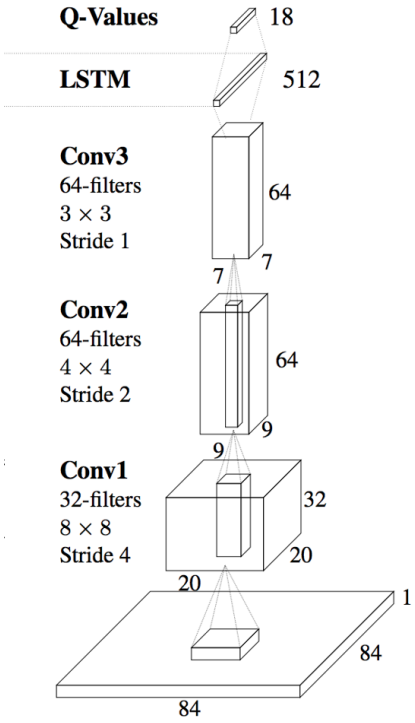


Figure 2.25: Deep Recurrent Q-Network Architecture

All previously proposed solutions have been tested on Atari games and they have performed well and often outperformed humans. The only data that was required was the raw pixels to make their decisions. The main common factor among most of those games was the fact that they are taking place in a 2D environments which are considered fully observable to the agents trying to solve them. In [45], they proposed a new learning technique to tackle a more challenging 3D environments in first-person shooter games. Such environments are more complex because of the partially observable states they involve. Typically, DRL techniques tend to utilize visual input as training data in building their models. However, the new proposed method improves that by enhancing these models to

take advantage of game features such as the existence of items or enemies, during the training phase. The developed model was trained to in addition to minimizing the Q-learning objective, it tries to learn those features simultaneously. This approach has shown a dramatic improvement in the training speed and the performance of the agent. Furthermore, the proposed architecture was designed to be modularized. This can allow to train different models independently regardless of the phase of the game. The proposed model was tested using ViZDoom environment and it was shown that its performance substantially outperforms built-in AI agents of the ViZDoom game as well as humans in certain scenarios. Another solution that tried to solve such challenging environment was the one proposed by [46]. Their solution was to use auto-encoders to convert high-dimensional imagery inputs into lower-dimensional representation and then used covariance matrix adaptation evaluation strategy (CMA-ES) to train a neural network. This approach was then tested on VizDoom and has shown promising results.

DQN algorithm that was developed by [3] was reliant on model-free RL. However, [47] claims that planning-based techniques can achieve higher results than the best model-free ones. Nevertheless, they suffer from two drawbacks. The first one is that they are considered to be slower than what needed for real-time play. Secondly, they exploit information that is unavailable to human players. In [47] the aim was to build a real-time player agent that can play Atari games better than DQN. The essential idea was to build a DL model capable of playing in real-time by utilizing training data generated using slow planning-based agents.

The proposed algorithm showed promising results in outperforming DQN in Atari 2600 games. However, it was slower in making decisions.

Classical games have been also considered as a challenge to solve using DRL. Giraffe [48] is a chess engine that was developed using DRL. In this engine, they used deep neural networks to extract meaningful features that would allow the agent to play chess professionally. Their experiments have shown a good competitive performance against other popular engines that are heavily dependent on expert crafted features. The advantage of Giraffe is that the features were learned by the machine without any intervention from a human as hand-crafted rules and features. A more recent achievement by DeepMind is AlphaGo. A new intelligent algorithm that was able to master the game of Go which has been considered the most challenging classical game for AI [49]. Despite the simplicity of Go rules, actions and even the objective which is to capture and occupy as much territory more than the opponent as possible, yet the huge number of states and solutions made it difficult to solve by computer AIs. The number of states in the game exceeds 10^{170} different states. AlphaGo however was able to learn how to play this game and win 5-0 against the European Champion. Moreover, it was able to achieve a better breakthrough by defeating the world top Go player that is considered the best over the past decade by winning 4 matches out of 5. The secret behind AlphaGo algorithm was the way of combining several AI and ML algorithms and make them work together. [50] introduced a new approach that merges DL and DRL with Monte Carlo Tree Search (MCTS) which allowed the

agent to learn the best policies in playing Go. Monte Carlo algorithm is a tree traversal search algorithm in its core. With the game states represented as a tree, the goal of the algorithm is to build a tree of possible future states that would eventually guide the agent equipped with MCTS to what actions to take. In order to build such tree, the algorithm would follow certain steps to do that. Initially the tree would be built up of only one node which is the root that represents the initial state. The agent would start by selecting the last known node laying in the leaves which is the current state and then tries to explore new ones. It selects an action for each new unknown node based on some exploration-exploitation strategy and then run parallel simulations called rollouts starting from each unknown node after taking the selected action by selecting random actions or following a rollout policy in order to expand the tree and learn the rewards that would be achieved from following those paths. Once it reaches a terminal state, it returns all learned states along with their Q-values and updates to the tree. The agent would then select the best action to take from the current state to follow according to the Q-values of available actions then the cycle repeats again [10]. However in AlphaGo, MCTS was modified to be more efficient in selecting what actions to take. They trained a deep neural network using supervised learning and called it SL policy network. The network was trained using Go expert games data to predict the probability of selecting actions in a given state by an expert human player. The aim of this network was to be used to select actions in MCTS from the current state[51]. Furthermore, they trained a second simpler network

using the same data to be used as a rollout policy. However, instead of relying on SL policy network resulted from the training phase, they used the parameters of the trained network to initialize a DQN model and trained it using self-play (which means playing against a random weaker version of the AI itself) to predict better policies which they called it RL policy network. Moreover, during the self-play phase, they trained another DL network named Value network that would predict the outcome from being in a certain state. They used the values predicted by this network to update the values of newly visited states during rollouts in MCTS in addition to the values returned from the rollout itself[50]. The algorithm has shown a huge improvement over available Go AI players, it was able to achieve a winning rate of 99.8% against them in addition to being able to defeat the top two human players in the world.

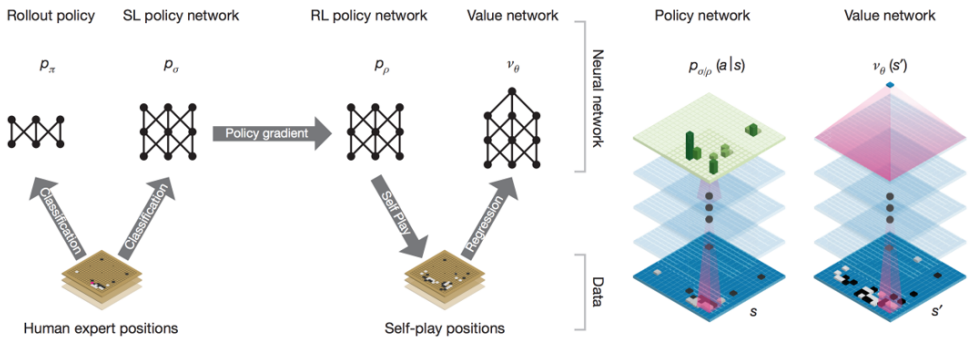


Figure 2.26: AlphaGo Networks

In opposite to DQN where it solely relies on rewards provided by the environment that are considered extrinsic to the agent, [52] proposed an agent that can develop self-motivation and curiosity to explore new states. This curiosity in

opposite to the rewards, can be thought of as an intrinsic reward signal generated by the agent itself. The motivation behind developing agents with intrinsic reward signal and curiosity-driven exploration ability is the extreme sparsity or absence of extrinsic reward signal that the agent might face in real-world scenarios. The suggested way to calculate this intrinsic reward was to incorporate the error in predicting the consequences of taken actions by the agent. Predicting the consequences of taken actions can be achieved by developing the ability to predict the next state given the current state and the taken action. However, the focus was only on the consequences that are generated by the agent's actions or would affect the agent while ignoring the rest. In order to achieve this, [52] proposed to encode the states into a new feature space that would focus on these certain consequences. This feature space is learned by modeling the inverse dynamics of the environment using self-supervised neural network. The Intrinsic Curiosity Module (ICM) consists of two modules in addition to the policy module that can be found in DQN that is used to predict Q-values given the current state as raw pixels. The first module is used to learn the new feature space and consists of two sub-modules. The first one encodes states of raw pixels into a new feature vector while the second one is used to model the inverse dynamics of the environment. This is achieved by taking as input the feature vectors of the current and next states and giving as an output the action that has been taken to lead to this result. The second module in ICM, models the forward dynamics by taking as input an action and feature vector of a state and predicts the feature vector that

encodes the next state. The intrinsic reward is then computed using the error in prediction given by the second module. The developed agent was tested in VizDoom environment in addition to Super Mario Bros. Furthermore, it was tested with three different setups. The first one with sparse extrinsic rewards while the second was without providing the agent with any extrinsic rewards pushing it to rely on its intrinsic motivation to learn good policies and the third was to train the agent in one level of a game and then test it in another one trying to measure its ability to generalize learned policies. The proposed model has shown in fact an improvement in learned policies by the agent even in the absence of extrinsic reward.

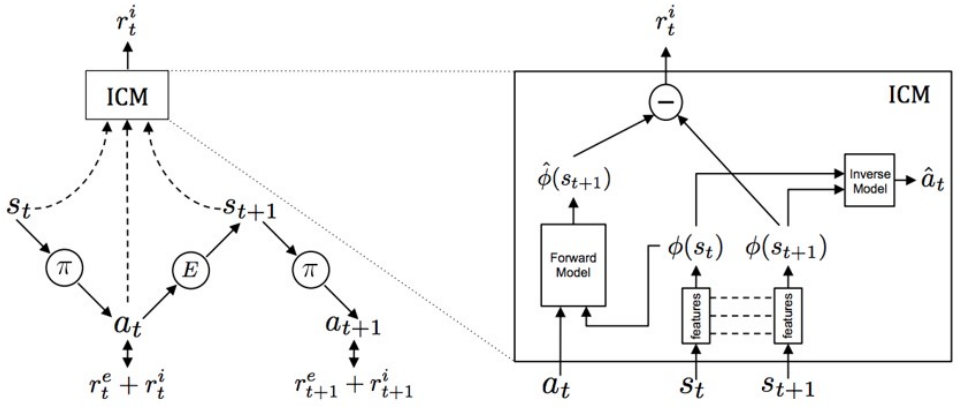


Figure 2.27: The Intrinsic Curiosity Module

CHAPTER 3

SIMPLIFIED DEEP Q-NETWORK

The ultimate goal of this work was to develop a technique that can help in reducing the complexity of DRL algorithms. This simplification however, has to maintain the accuracy of the developed models such that even though it reduces the over all training time and the processing power required yet it preserves the performance of the trained models if not enhancing them. There is a variety of DRL algorithms in literature as was discussed in section 2.4. They all try to develop intelligent agents that would be able to solve complex problems represented by games without any human intervention. This complexity comes from the fact that the only input provided to the agent is high-dimensional sensory input represented by the frames of the game it tries to solve. These algorithms and in order to give the developed agents the ability to make sense of provided inputs, they tend to rely on CNNs. CNNs are well-known to be the best algorithms in handling images and extracting

meaningful features from them as was shown in section 2.2.4. When used in DRL algorithms, they are trained to act as value function approximators in order to guide the agent to what actions to take and that is defining its behaviour when solving the games. However, the architecture of the used CNNs tend to be complex due to the complexity of the games the agent tries to solve. The more complex the architecture of CNNs is, the more sophisticated the extracted features are and this can in fact help the agent in solving those complex games. However, this sophistication increases the demand on processing power and it further increases the time required to train. Therefore, it makes sense to start the simplification process by first trying to reduce the complexity of the architecture of CNNs used in DRL. The focus of this work is on DQN which is the baseline algorithm in DRL and many algorithms are considered to be enhancements over this algorithm. However, before discussing how the CNNs can be simplified, it is essential to discuss one of the reasons for making them complex in the first place.

3.1 Observability Issue

If the games from ALE or even ViZDoom used as environments in MDP, then those games can be considered partially observable if the input given to the agent from the environment is only one single game frame at a time. If we take the pong game for example, one frame can be ambiguous as it appears in Figure 3.1. It does not give any clue neither about the direction of the ball movement nor about the speed of the ball. This indeed makes it difficult on the agent to infer

useful information and develop good behaviors because the same frame can have different meanings in non-similar situations. Thus, it will make it difficult on the agent to fully understand the situation it is facing and it will over complicate the training process. In the literature, this was solved using two different approaches. The first one was used in DQN ([3, 40]) while the second was introduced in DRQN ([44]). The way DQN solved the problem was to consider the state in the environment to be a sequence of observations (frames) and the taken actions. This was reflected in the CNN as an input of stack of four frames. When the CNN is fed with the stack of frames it can deduce the movement and the speed of the ball in Pong game for example making it fully observable. On the other hand, DRQN approach to solve this challenge was to use LSTM which has the ability to maintain previous experiences as a sort of memory. However, both techniques are considered to be complex and demanding in terms of processing power. The stack of frames complicate the architecture and LSTM tends to converge slowly.

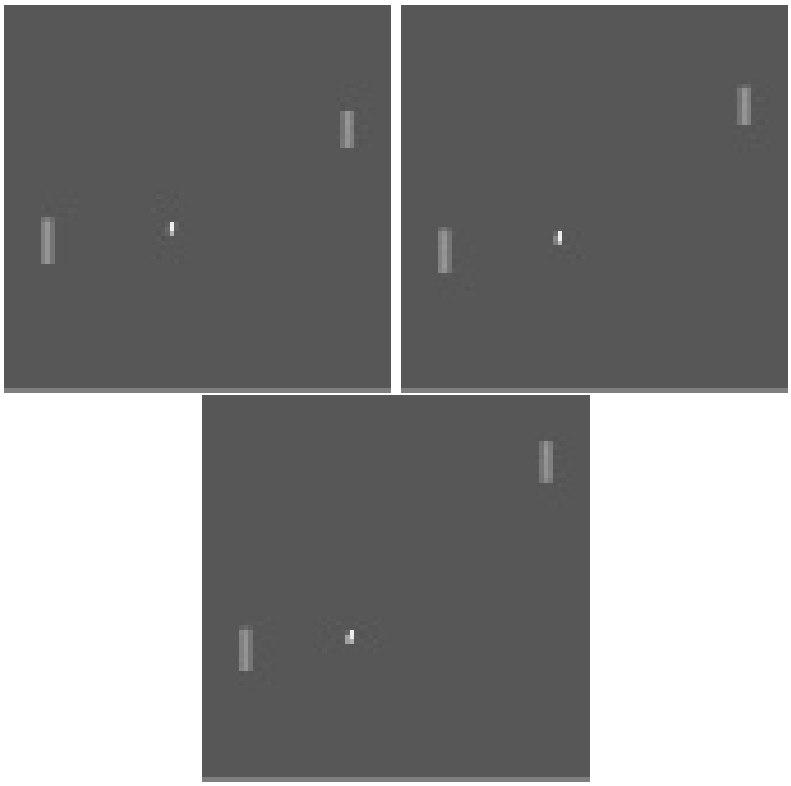


Figure 3.1: State Ambiguity

3.2 The Merging Technique

In this thesis, a third technique is proposed that can be as effective as the stack of frames, yet it requires simpler CNN. The main advantage of the stack of frames is to give the CNN enough data frames that can be used to infer useful information about the state. Therefore the objective is to replace the stack of frames with a simpler solution that can produce the same information. The idea of the proposed technique is based on merging the stack of frames in one frame before passing it to the CNN. However, by only merging the frames without any preprocessing it can cause the same effect as using one frame or even worse. If the merge consists of four frames it would look like Figure 3.2 causing big ambiguity.

The agent will again be unable to infer any information about the movement direction and speed of ball and hence it will become again partially observable. The key element in eliminating the ambiguity using the merging technique is to reduce the intensity of the frames as they get older. This will give the agent a sense of how fast the ball is moving and in which direction in Pong game for example using the features learned in the CNN. The direction of movement can be learned by following the direction opposite to the fading-out and the spaces between the balls positions indicate the speed as it is shown in Figure 3.3.

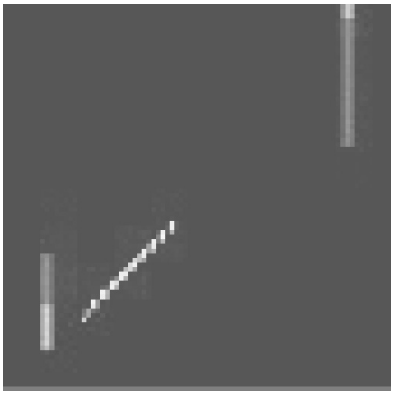


Figure 3.2: Merging Frames in Pong Game

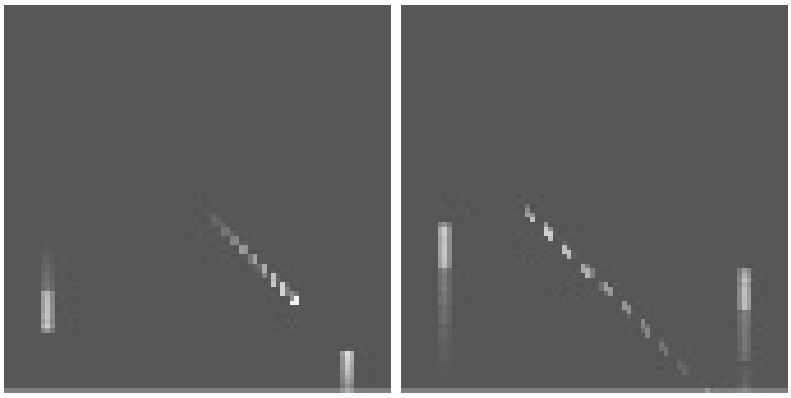


Figure 3.3: Direction & Speed of The Ball in Pong Game

The technique requires the agent to keep a copy of the last n frames (depending on the size of the history) and then merge them together in a fading-in fashion by incrementally increasing the intensity of the frames until it reaches to the most recent one with full intensity as can be seen in Figure 3.4 for Pong game and Figures 3.5 and 3.6 for ViZDoom. Once the frame generated from merging the old frames is ready it would be fed to the CNN for value approximation. One of the advantages of this technique over stack of frames DQN is that the number of frames used in the merge can be increased beyond four frames without affecting neither the architecture of the CNN nor the performance.

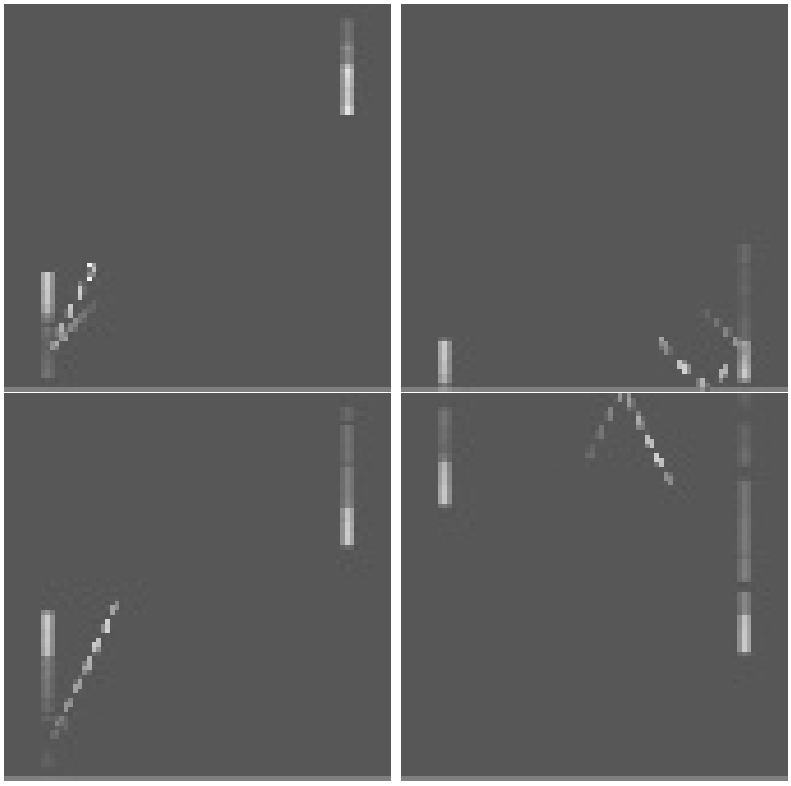


Figure 3.4: Merging Technique in Pong Game

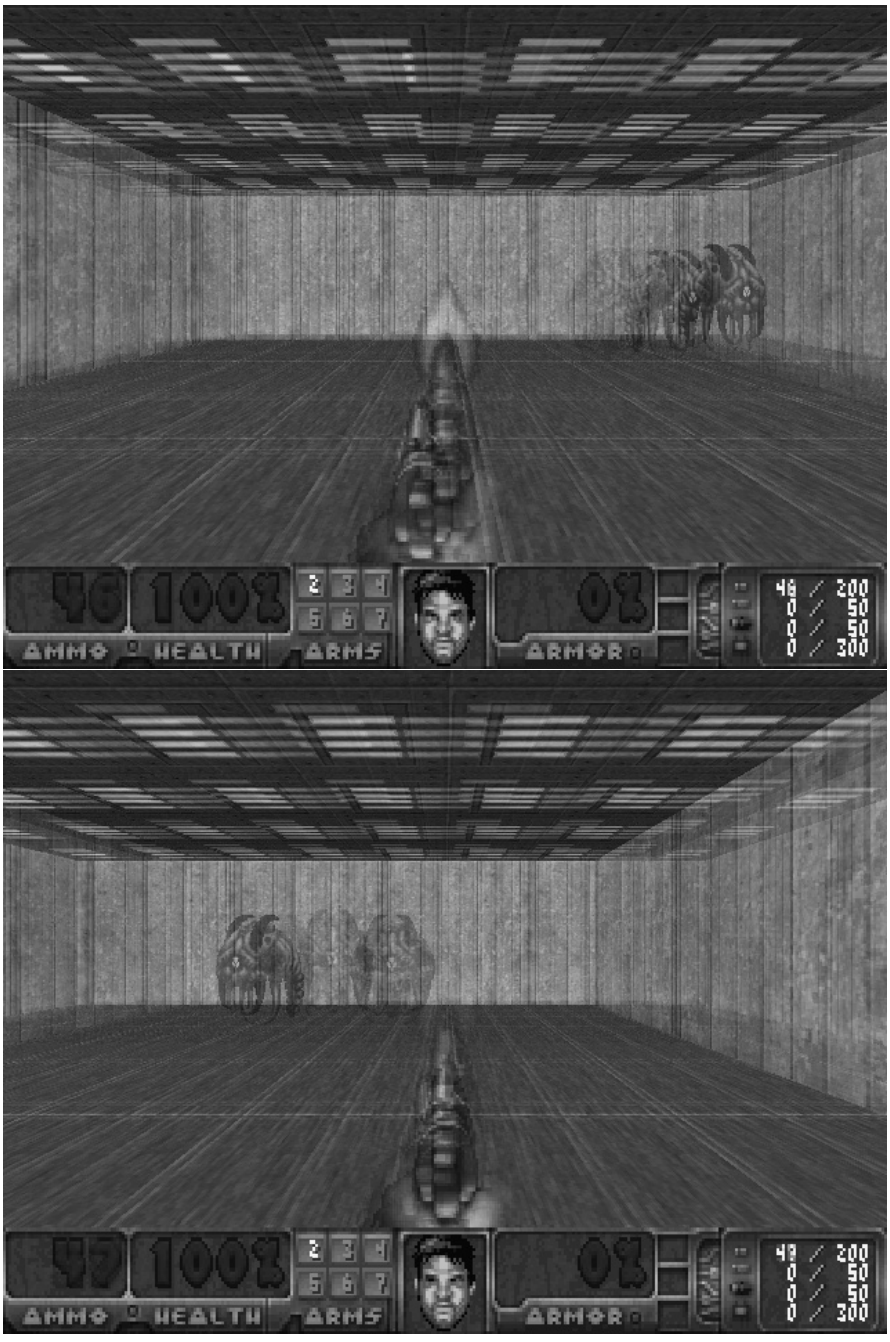


Figure 3.5: Merging Technique in ViZDoom - Basic Scenario

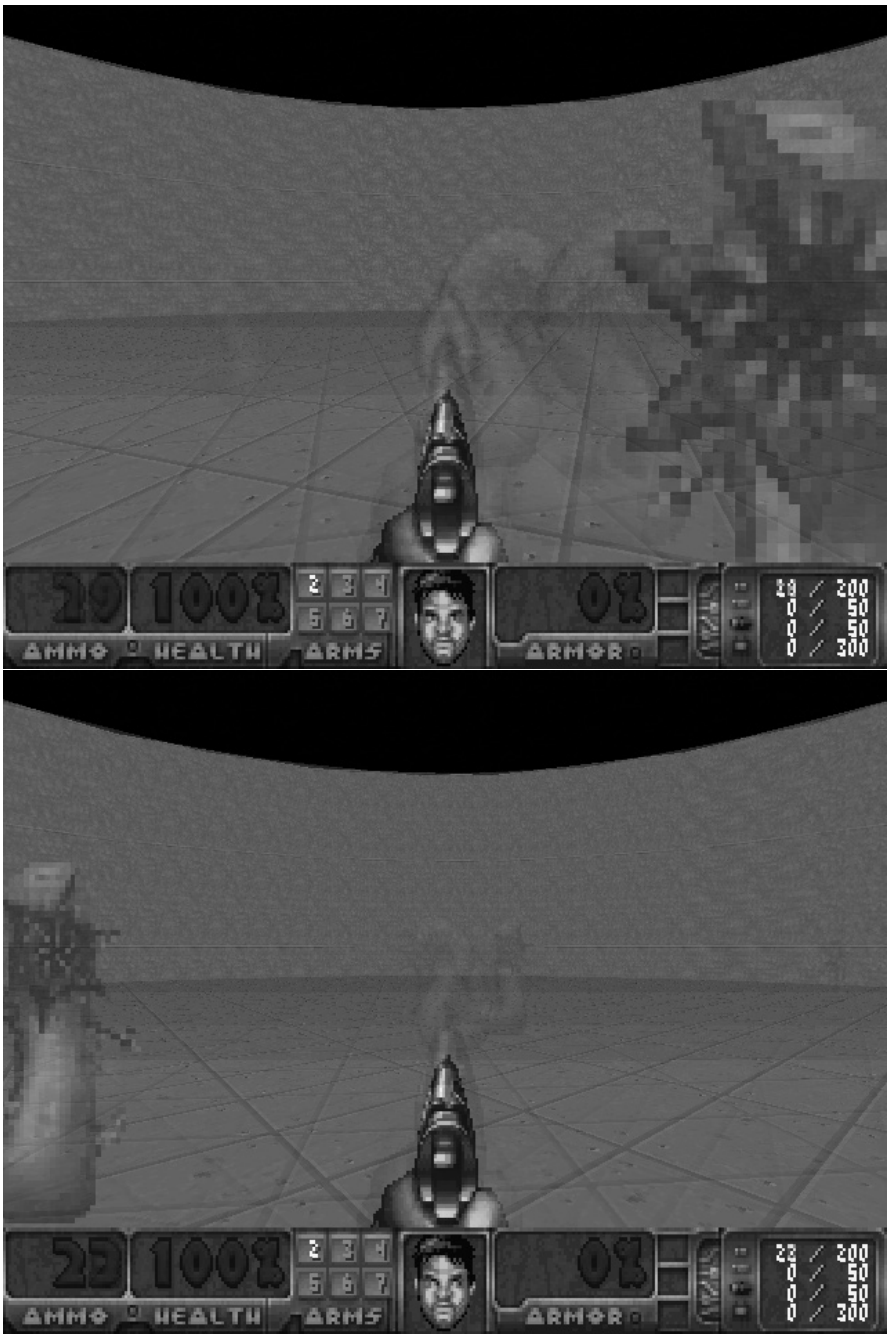


Figure 3.6: Merging Technique in ViZDoom - Defend The Center Scenario

3.3 Background Removal

To further improve the performance of DQN when equipped with the merging technique, another preprocessing step was implemented. Previously, DQN with the stack of frames, it relied on four separate frames for approximating the value function. Each one of these frames was in full intensity. Therefore, the information presented in each one of the frames maintained its strength and effect. However, in the new presented technique, the information embedded within the frames loses its strength because of the intensity reduction. Furthermore, the background color in the frame is causing the information to fade as the data moves deeper in the CNN and hence reducing the effectiveness of the frame merging technique. In order to fix this issue, a closer look is needed into how frames are represented. The representation of frames in computers appears as arrays of numbers where each pixel is represented in a cell. Since all the frames used in DQN are in gray scale then each cell in the array can have a value between 0 and 255 where the first value represents black while the other value represents the white color and the shades of gray are represented by the values in between. Hence, the background in the merged frames can impact the extracted features because of its grey color. Therefore, in order to overcome this issue, the background must be eliminated from any calculations that can take place in the CNN. This can be accomplished using background removal that would only leave the relevant information in the frame. In Pong game for example the only thing to be left are the two paddles and the ball as shown in Figure 3.9.

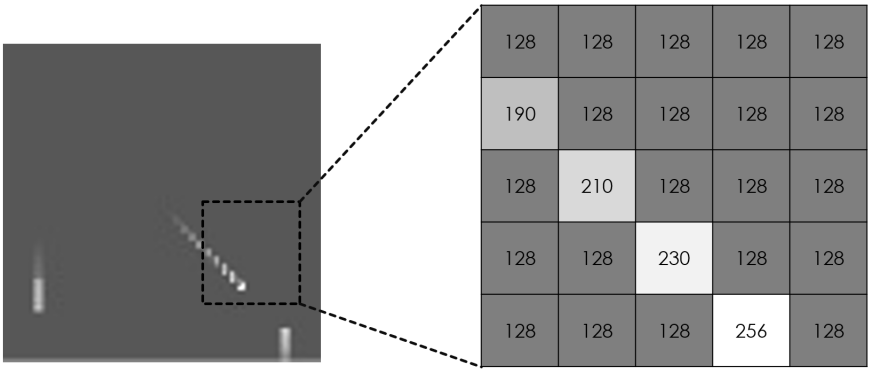


Figure 3.7: Pong Frame - With Background

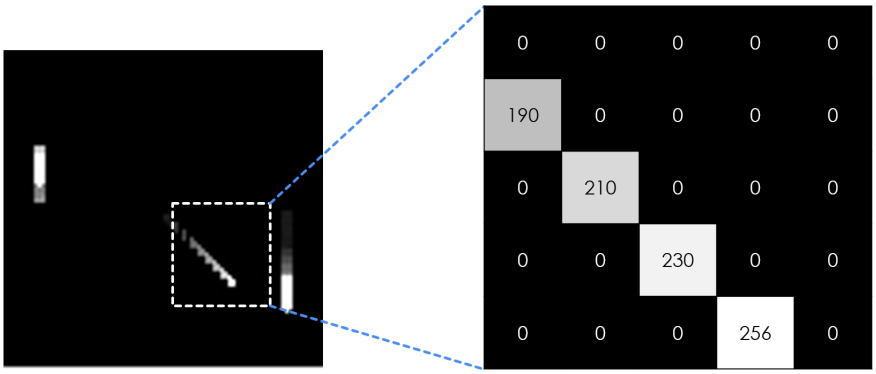


Figure 3.8: Pong Frame - Without Background

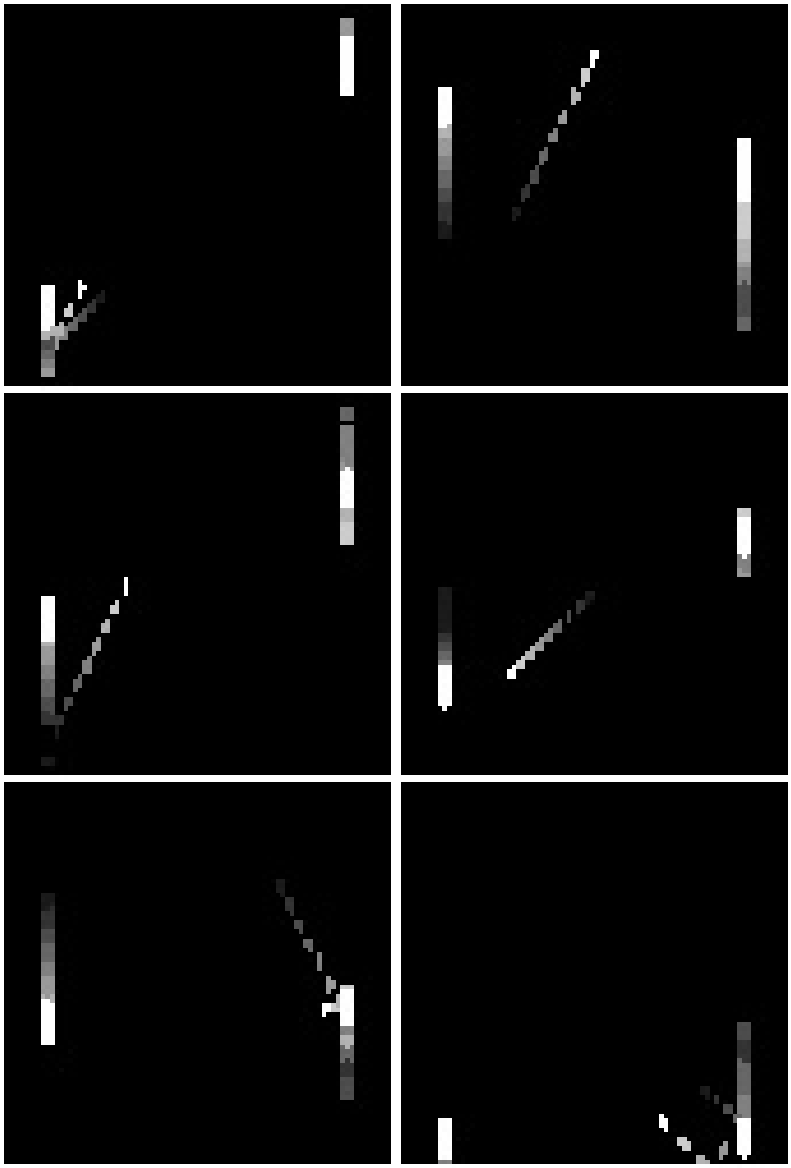


Figure 3.9: Merging technique with background removal

CHAPTER 4

EXPERIMENTS

4.1 Experiments Setup

All the experiments were conducted on a Linux machine running Ubuntu OS v16.04LTS with the following specifications:

- Intel i7-6900K Octa-core @ 3.2 GHz
- 64 GB RAM
- Nvidia Geforce GTX 1080 - 8 GB
- 500 GB Samsung 850 EVO SSD

The implementations of the algorithms were built using Python 2.7 in addition to Python 3.5. In both Python versions the following libraries were used as part of the implementation:

- TensorFlow 1.1.0

  TensorFlow is an open-source library that was developed by Google Brain

researchers and engineers. It is mainly used in ML applications and research. It is the core library that was used in building all the algorithms that were tested in this research. It contributed in building both the DL part in addition to the RL one. One of the main features of TensorFlow is its ability to run on GPUs utilizing Nvidia CUDA and CuDDN [53] for accelerated performance.

- Numpy 1.13

  Numpy is a scientific computational package that is known for its ability of handling multi-dimensional arrays efficiently in Python in addition to other numerical functions.

- OpenCV v3.2

  OpenCV is considered as one of the well-known and leading libraries in computer vision and image processing. It was used in the research as part of the implementation of the merging technique.

- Gym 0.9.4

  Gym [54] is an open-source toolkit that was developed by OpenAI that offers a standardized set of environments for testing and comparing RL algorithms. In this research the ALE [32] was used in testing and comparing the algorithms. In particular, Pong game was the testing environment from ALE that was used in some of the conducted experiments.

- ViZDoom [33] is an open-source RL tool that provides Doom game inspired

challenges for AI bots.

- Pandas 0.21 is an open-source high performance data structure library for Python programming language.
- Matplotlib 2.0 is a 2D plotting library for Python programming language.

4.2 Experiments Design

The main purpose of the experiments was to test if using the merging technique can in fact reduce the training time needed to train DQN when compared with the stack of frames and how can this modification affect the performance of the developed agent.

To ensure that the merging technique is implementation independent, it was tested using two different implementations of DQN. The first one was built using Python 2.7 and was based on an open-source skeleton implementation of DQN [55]. This implementation was missing the core parts of DQN and only offered a guidance and some utility classes and functions that helped in memory management. The rest of the functionalists along with the core ones were implemented in order to have a fully functional DQN. The second implementation of DQN was based on the DQN baseline offered by OpenAI [56] that was built using Python 3.5. The advantage of using well-known and open-source baseline implementations is to help researchers in reproducing the results presented in this research.
Even though two different implementations were used in the experiments however,

both implementations shared the same CNN architecture. Moreover, they both applied the same preprocessing step on the game frames.

Regarding the architecture of the CNN, it consists of three hidden convolutional layers followed by two fully connected ones. The first layer has 32 filters of size 8x8 and a stride of 4 while the second layer has 64 filters of size 4x4 with stride of 2 and finally the third layer has 64 filters of size 3x3 and a stride of 1. All three layers were followed by rectifier nonlinearity units. The first fully connected layer consists of 512 rectifier units while the output layer which is also fully connected has a single output for each action in the game. In the case of Pong, there were six different actions and in case of ViZDoom it was a combination of eight actions. The input that is fed to the CNN is an 84x84 gray scale image produced by the preprocessing step in case of Pong game and 96x48 in case of ViZDoom. However, in the case of DQN with the stack of frames implementation this input becomes 84x84x4/96x48x4 tensor. On the other hand, it remains 84x84x1/96x48x1 when the merging technique is used with further preprocessing steps applied to the image before feeding it to the network. Instead of using RMSProp as in the original DQN, ADAM algorithm was used because of its advantages.

The differences between the implementations can be enclosed in the following: The first difference was in the implementation of experience reply buffer and the memory handling. The second difference is in the learning rate used in ADAM technique. In the first implementation of DQN the learning rate had two values. It starts with the value of $1 \times e^{-4}$ in the first half of the experiment. Then it

reduces it to $5 \times e^{-5}$ in the second half. However, in OpenAI baseline DQN the learning rate remains $1 \times e^{-4}$ for the whole experiment. Finally, they differ in the exploration policy followed by the agent by using different ϵ values such that the agent developed using the first implementation tends to be more explorer than the one developed by the other one.

The experiments were conducted to help answering the following questions:

- What is the time required to train DQN with a stack of frames in comparison to one frame only and how can that impact the performance?
- In merging technique, how many frames are considered sufficient to outperform the stack of frames in DQN?
- Can the removal of the background from the frames improve the performance of the agent?

Furthermore, the experiments were later extended to include the modified algorithms of DQN that are DDQN and Dueling DQN available in OpenAI baselines to test if the merging technique can have the same impact on them as it has on DQN.

All three techniques were tested to learn Pong game from ALE. However, to further test the effectiveness of the merging technique on more complex frames than the ones presented in Atari games, DQN was tested on two scenarios from ViZDoom benchmark tool.

4.3 Experiments Results

In this section, the conducted experiments will be discussed in details along with their results.

4.3.1 Experiments I

This set of experiments was conducted using the DQN implementation that is based on the skeleton DQN. Pong game from ALE was used in these experiments to train the agents.

Experiment I.I

The first experiment in this set was to test the impact of replacing the stack of frames with one frame only without any frame merging. By referring to Figure 4.1, it is clear that reducing the stack to one frame can cause a huge degradation in performance. However, Figure 4.2 shows that the required training time has been reduced by around 16%. This means that if we can improve the performance of the one frame model we can have a faster model and this can be achieved by applying merging technique.

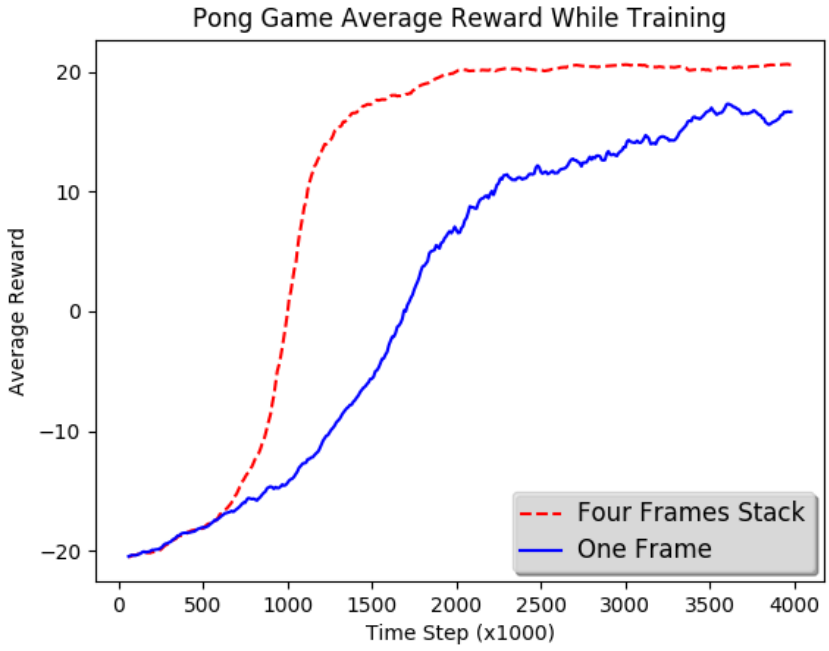


Figure 4.1: Experiment I.I - Game Average Rewards - DQN (Stack vs One Frame)

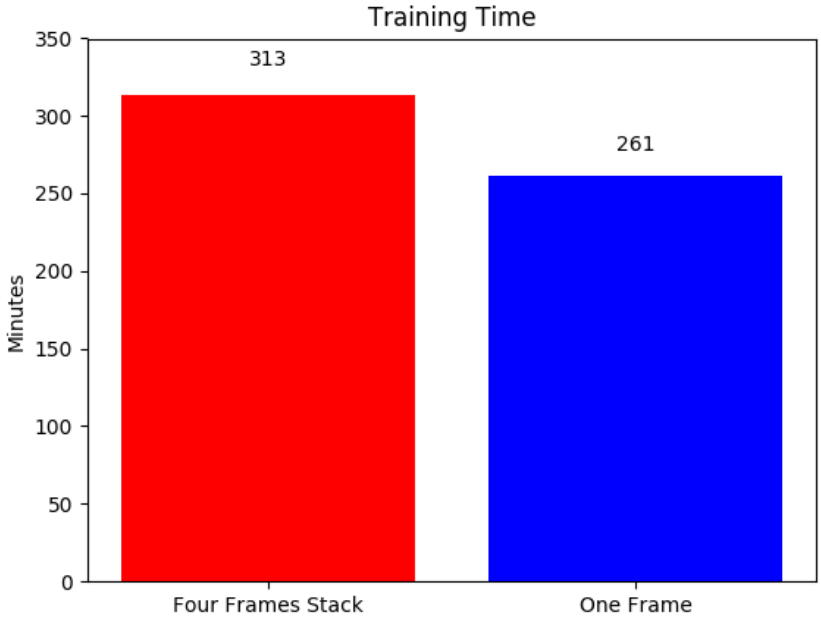


Figure 4.2: Experiment I.I - Training Time - DQN (Stack vs One Frame)

Experiment I.II

In this experiment, the aim was to test the performance of DQN when supplied with merged frames. The experiment tries to identify the impact of merging different numbers of frames on both performance and training speed to find the best number of frames to be merged.

Figure 4.3 shows that merging four and six frames did not produce good results. However, when merging ten frames the results were promising yet it was still less than the original DQN (Figure 4.6). However, merging the frames have shown an improved performance over one frame while maintaining the less training time than DQN by 10% even when merging ten frames as shown in Figure 4.7.

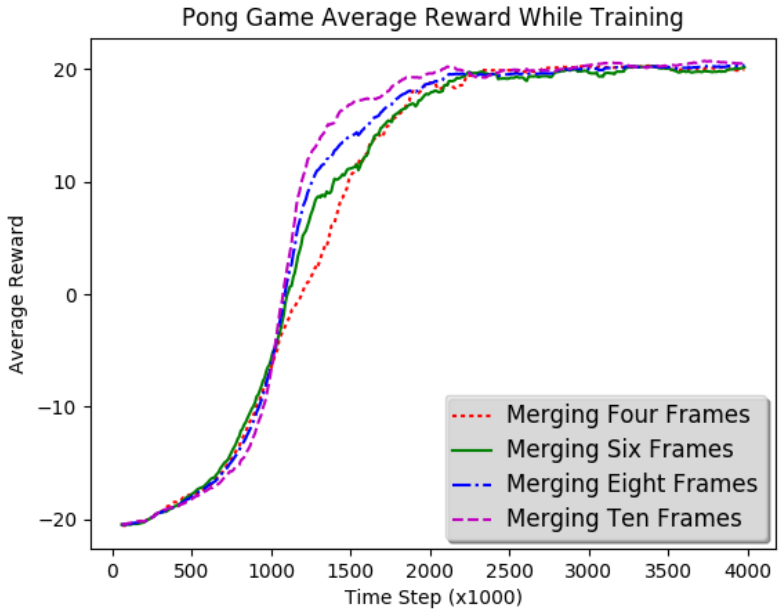


Figure 4.3: Experiment I.II - Game Average Rewards - DQN (Merging Multiple Frames)

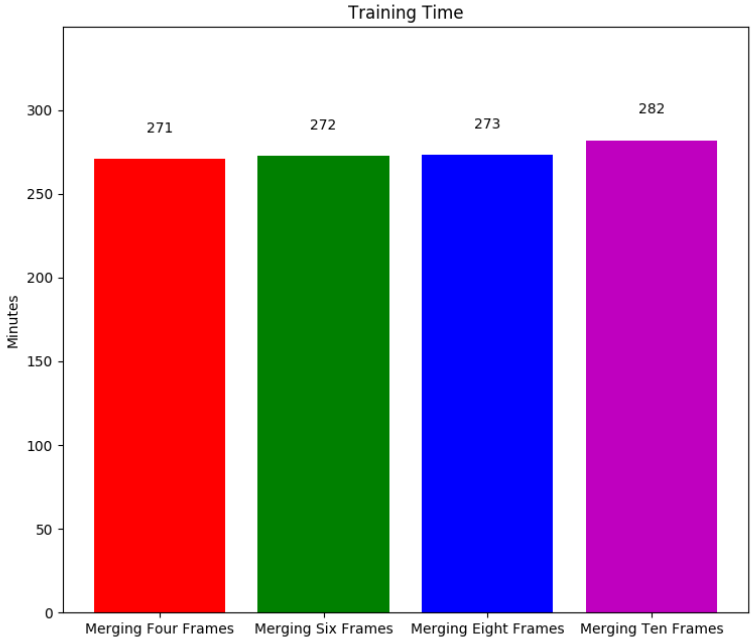


Figure 4.4: Experiment I.II - Training Time - DQN (Merging Multiple Frames)

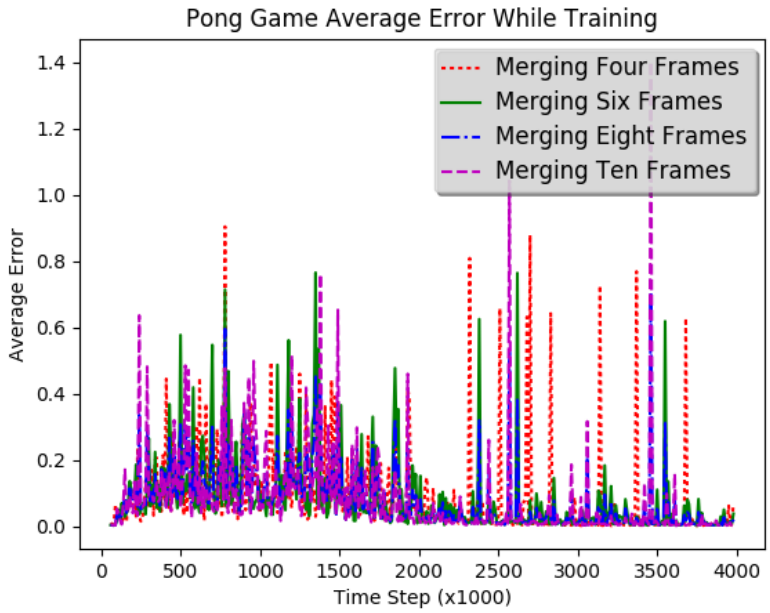


Figure 4.5: Experiment I.II - Average Errors - DQN (Merging Multiple Frames)

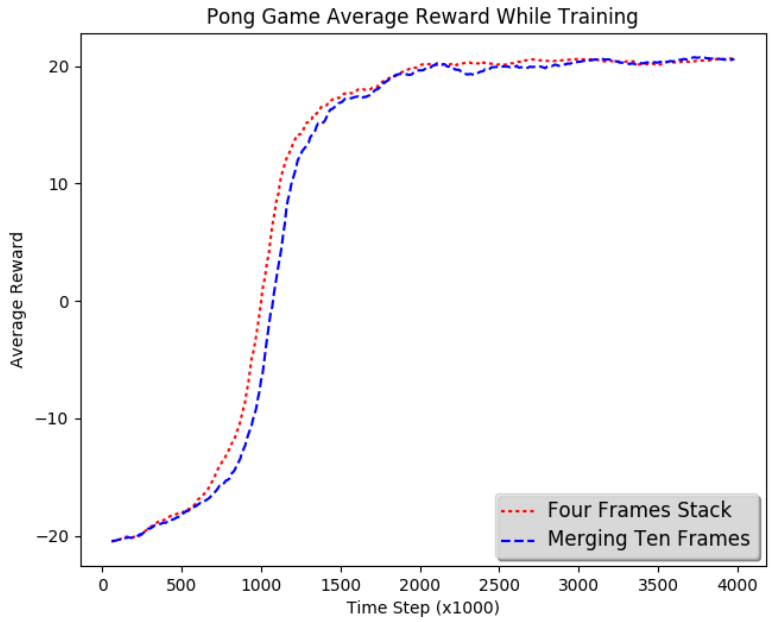


Figure 4.6: Experiment I.II - Game Average Rewards - DQN (Stack vs Merging Ten Frames)

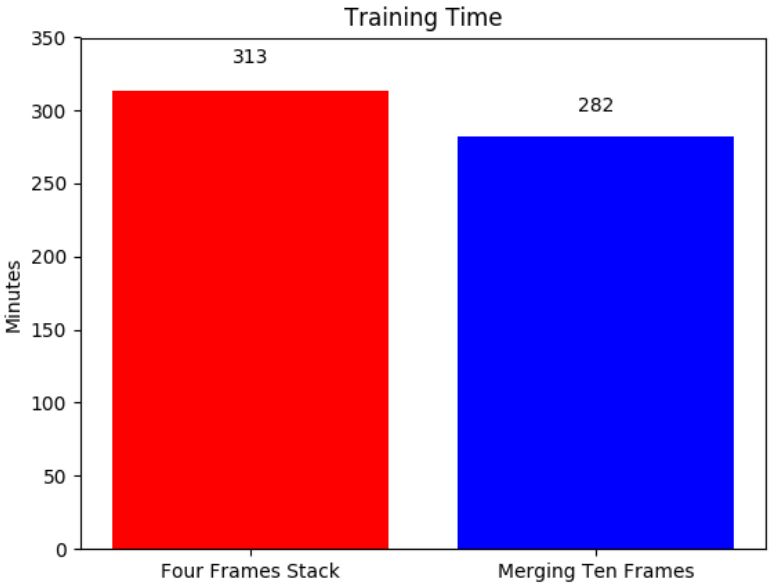


Figure 4.7: Experiment I.II - Training Time - DQN (Stack vs Merging Ten Frames)

Experiment I.III

As suggested in section 3.2, the results of merging frames technique can be improved by removing the background. In this experiment, the aim was to test for this hypothesis.

By referring to Figure 4.8, it can be seen that by removing the background, the performance when merging four frames without a background has improved to almost reach the performance when merging ten without a background. Furthermore, Figure 4.11 shows that merging ten frames has exceeded DQN slightly in performance. In terms of training time, Figure 4.12 shows that this improvement in performance did not cost any extra training time when compared with merging technique without background removal. Actually, removing the background of ten frames and merging them requires around 10% less time than DQN with its stack of frames and yet it slightly outperforms it.

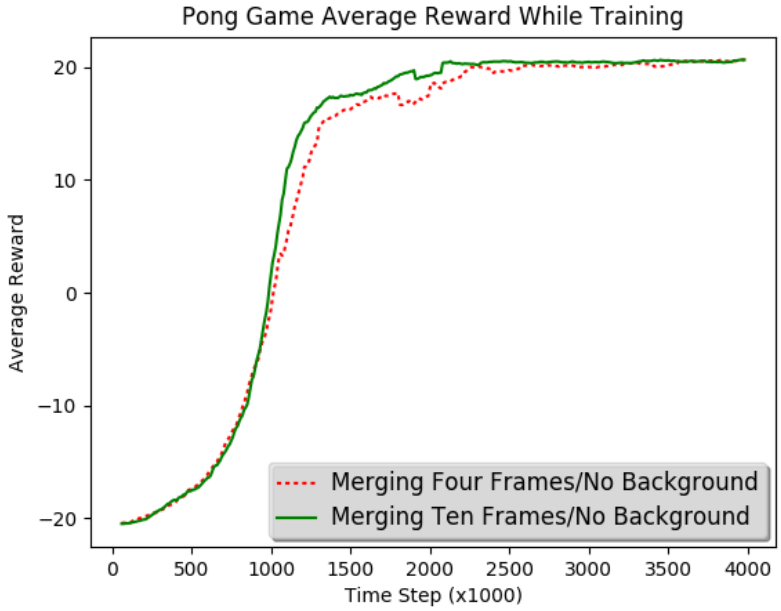


Figure 4.8: Experiment I.III - Game Average Rewards - DQN (Merging Four Frames Without Background vs Merging Ten Frames Without Background)

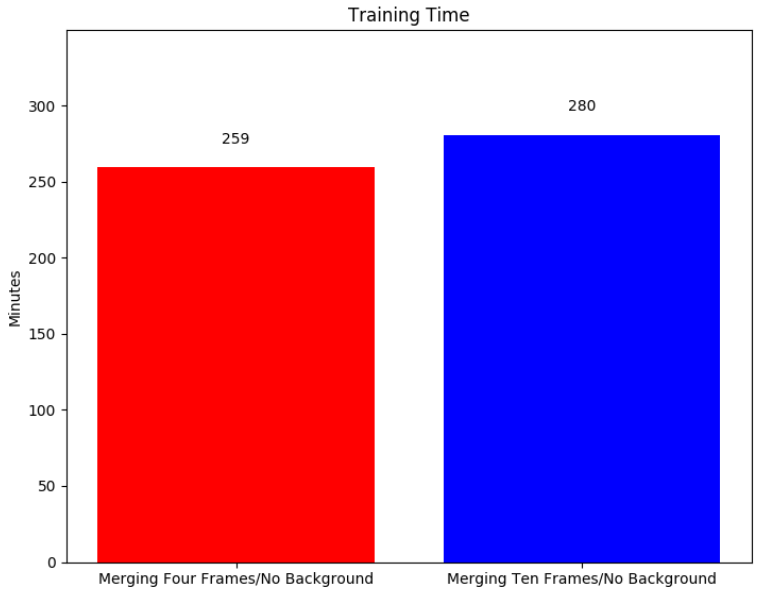


Figure 4.9: Experiment I.III - Training Time - DQN (Merging Four Frames Without Background vs Merging Ten Frames Without Background)

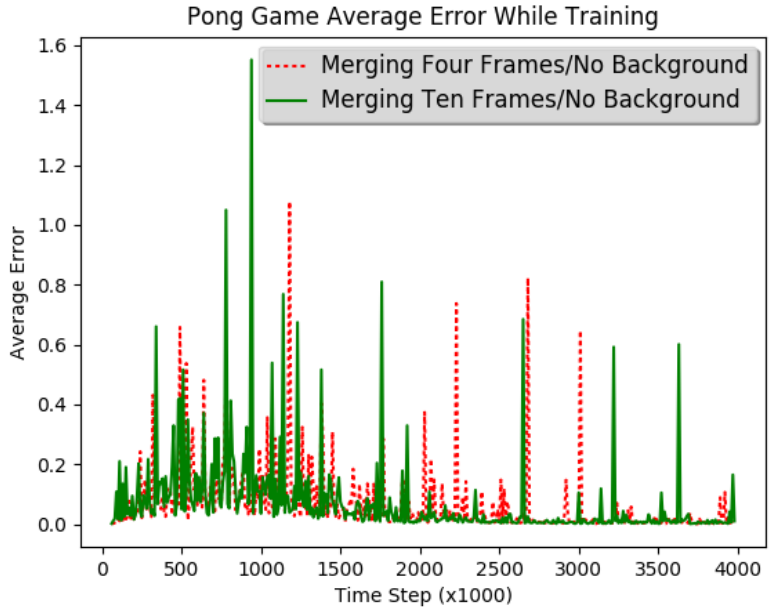


Figure 4.10: Experiment I.III - Average Errors - DQN (Merging Four Frames Without Background vs Merging Ten Frames Without Background)

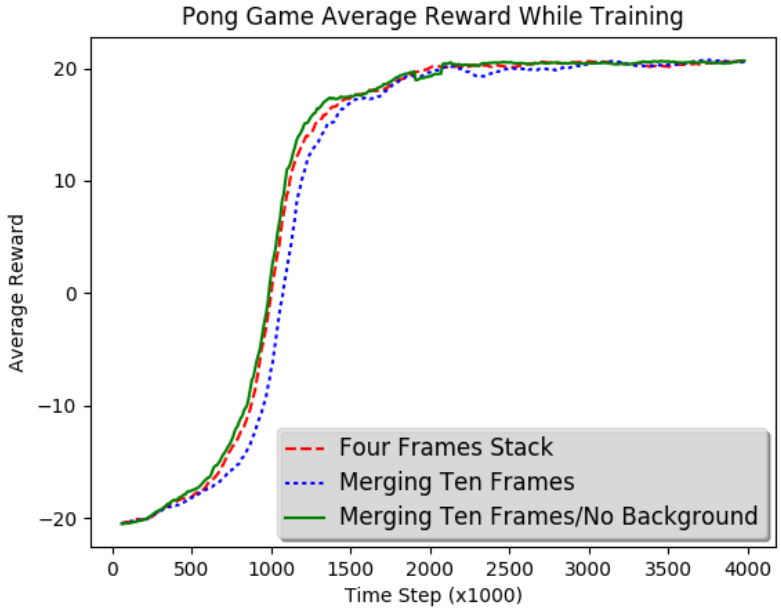


Figure 4.11: Experiment I.III - Game Average Rewards - DQN (Stack vs Merging Ten Frames vs Merging Ten Frames Without Background)

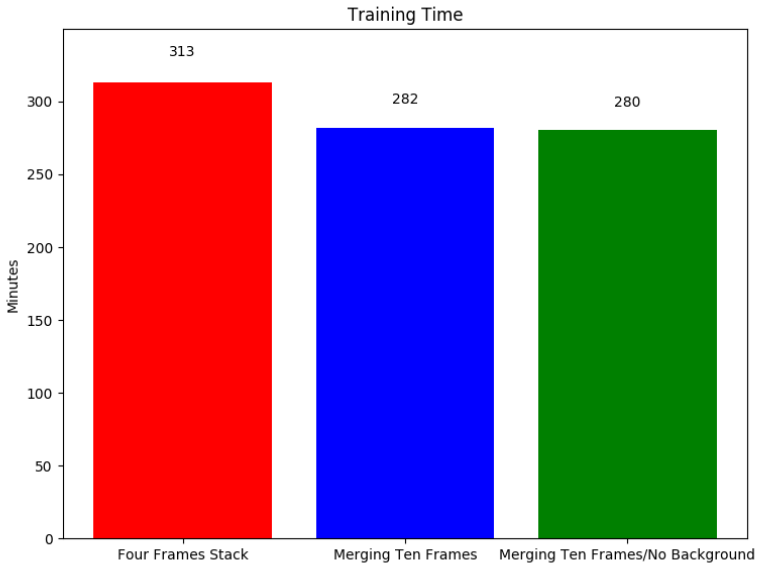


Figure 4.12: Experiment I.III - Training Time - DQN (Stack vs Merging Ten Frames vs Merging Ten Frames Without Background)

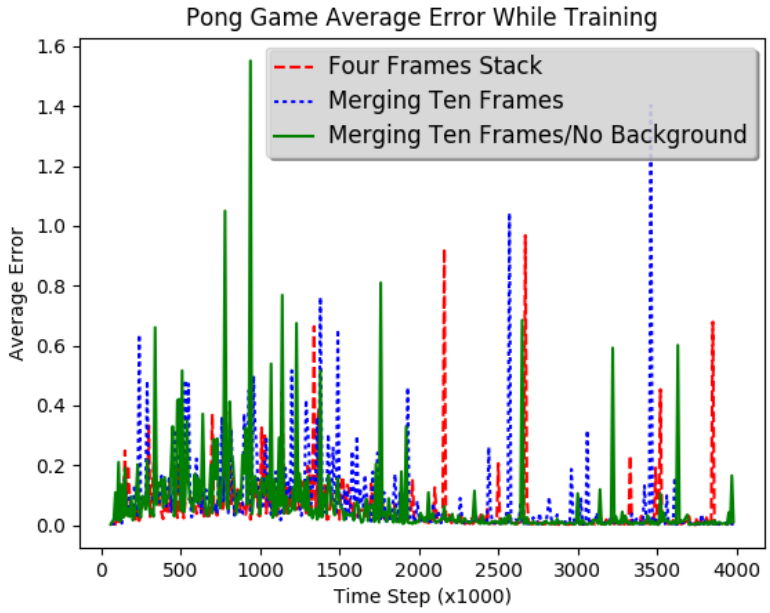


Figure 4.13: Experiment I.III - Average Errors - DQN (Stack vs Merging Ten Frames vs Merging Ten Frames Without Background)

4.3.2 Experiments II

This set of experiments was conducted using the OpenAI baseline DQN, DDQN and DuelingDQN implementations. The aim of this set of experiments is two-fold: the first half is to confirm that the improvements presented by the merging technique in terms of performance and training time is implementation independent by matching the results obtained from the corresponding experiments from the first set. Secondly, to test the impact of the merging technique on DDQN and DuelingDQN. The experiments conducted in this set used Pong game from ALE to train the different agents.

Experiment II.I

In this experiment, the goal was to test the performance of DQN with one frame when compared to DQN with a stack of four frames. Figure 4.14 shows again a degradation in performance when using one frame with DQN with the benefit of offering less training time by 23% (Figure 4.15). However, if the training time is compared to the first implementation of DQN with one frame, it can be noticed that the difference is huge (around 30% slower as shown in Table 4.1). This increment in training time is due to the different implementation that affect memory management and experience reply buffer.

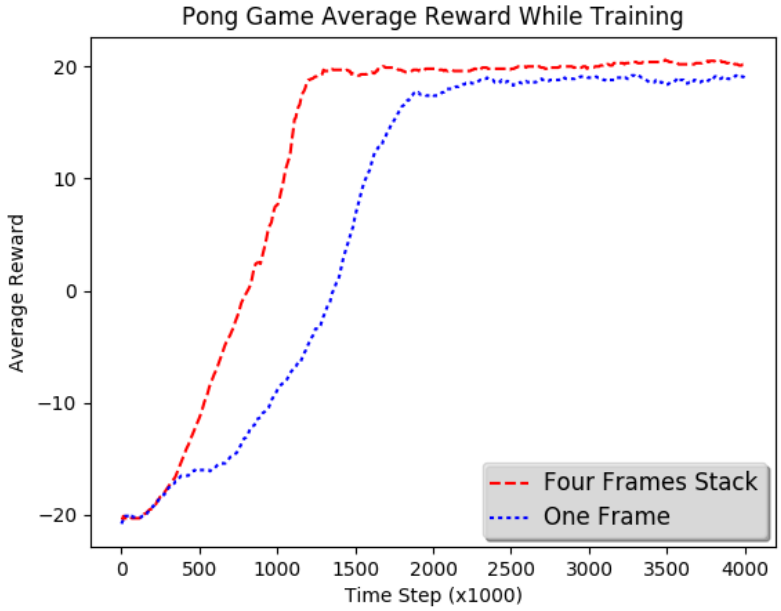


Figure 4.14: Experiment II.I - Game Average Rewards - DQN Baseline (Stack vs One Frame)

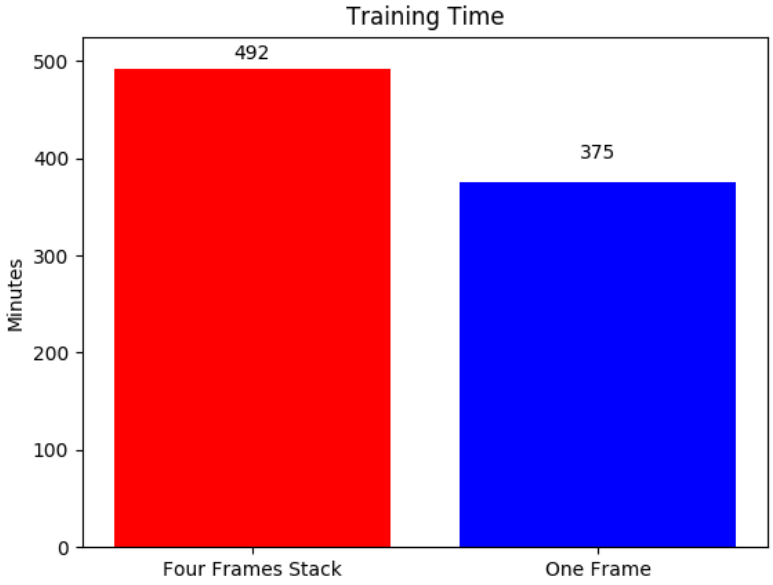


Figure 4.15: Experiment II.I - Training Time - DQN Baseline (Stack vs One Frame)

Experiment II.II

When comparing the effect of the number of merged frames on the performance of baseline DQN, Figure 4.16 shows that four and six frames performed poorly compared to ten frames in opposite to eight frames that showed very similar performance. If the results of this experiment compared with the results of experiment I.II, it can be seen that four and six frames performed similarly in both experiments. However, when it comes to eight frames, in the first experiment it showed less performance than the ten frames as compared to the similar performance in baseline DQN. This is due to the difference in the hyper-parameters in both implementations. However, in both cases, merging ten frames has shown the best performance and increasing the number of frames had slight impact on the training time.

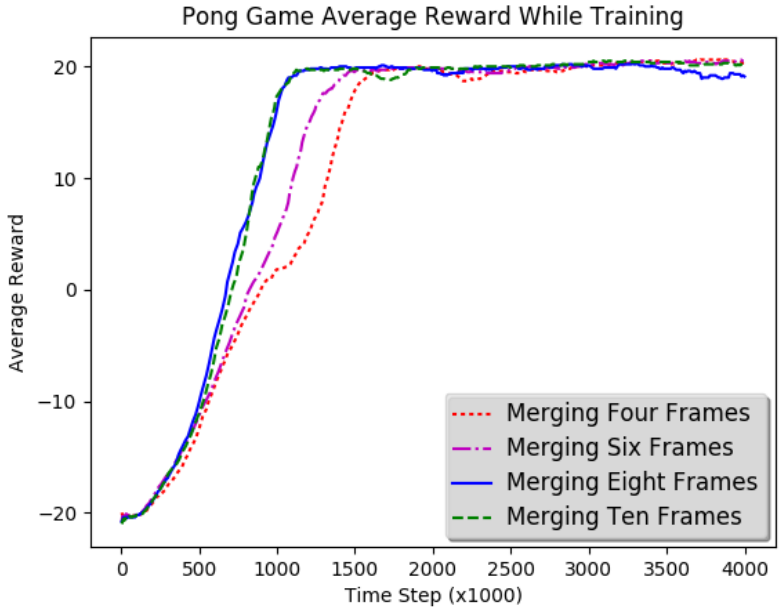


Figure 4.16: Experiment II.II - Game Average Rewards - DQN Baseline (Merging Multiple Frames)

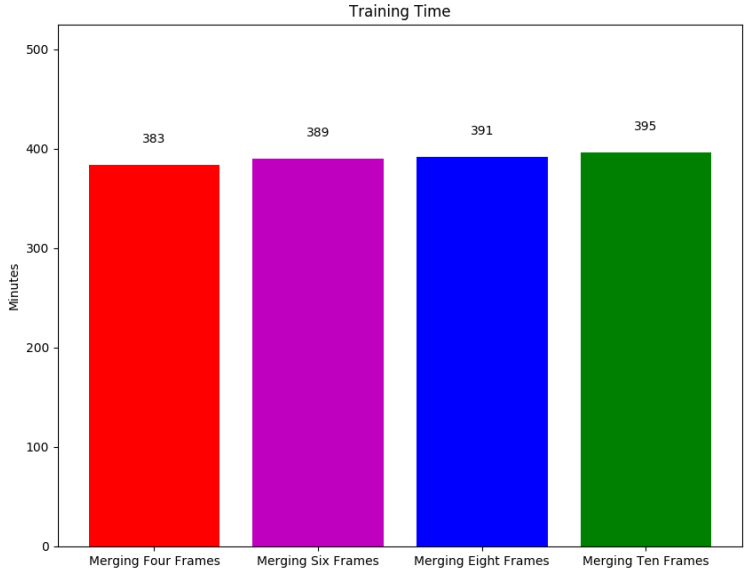


Figure 4.17: Experiment II.II - Training Time - DQN Baseline (Merging Multiple Frames)

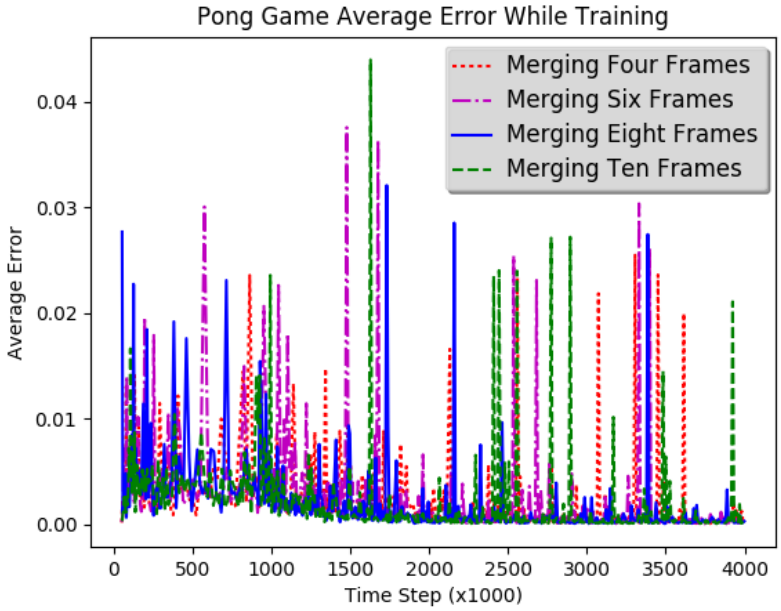


Figure 4.18: Experiment II.II - Average Errors - DQN Baseline (Merging Multiple Frames)

Experiment II.III

The hypothesis from section 3.2 that claims the improvement in performance by removing the background has again proven to be true when applied on baseline DQN. In addition to improving the performance of merging four frames to be very similar to ten frames (figure 4.19), it has improved the performance of baseline DQN with merged ten frames to outperform the stack version with a good margin (Figure 4.21). In fact, an interesting point to notice from Figure 4.21, is that the performance of merging ten frames has exceeded the stack version without even removing the background. This result is opposite to what was obtained in the experiments applied on the skeleton version of DQN. This improvement in performance in baseline DQN was accompanied with a reduction in the required

training time by around 20% for the merge of ten frames without removing the background and by 18% when the background is removed as shown in Figure 4.11.

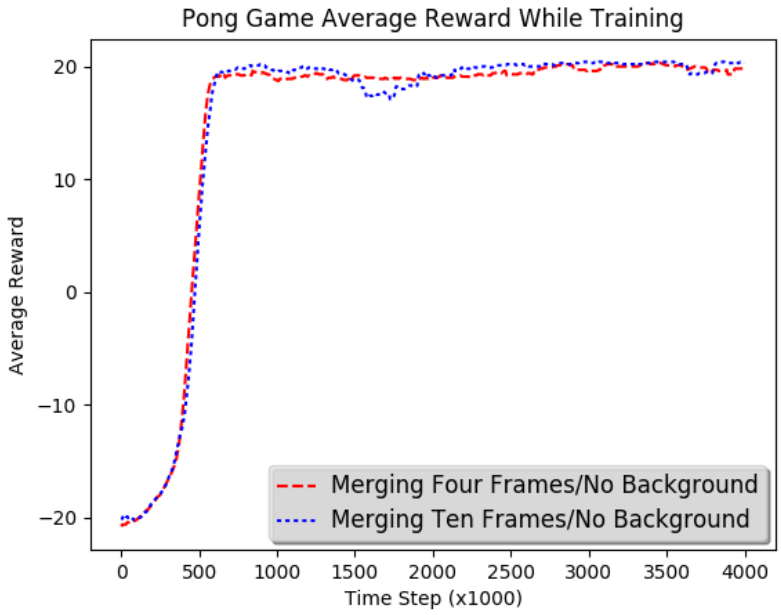


Figure 4.19: Experiment II.III - Game Average Rewards - DQN Baseline (Merging Four Frames Without Background vs Merging Ten Frames Without Background)

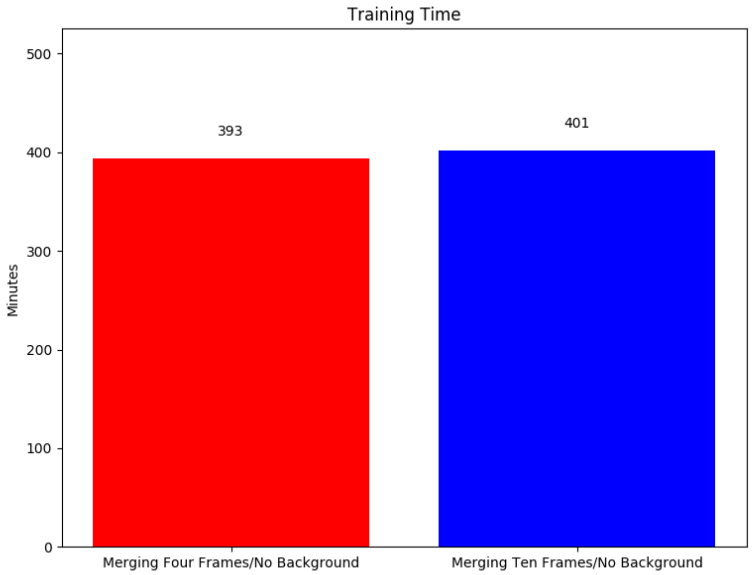


Figure 4.20: Experiment II.III - Training Time - DQN Baseline (Merging Four Frames Without Background vs Merging Ten Frames Without Background)

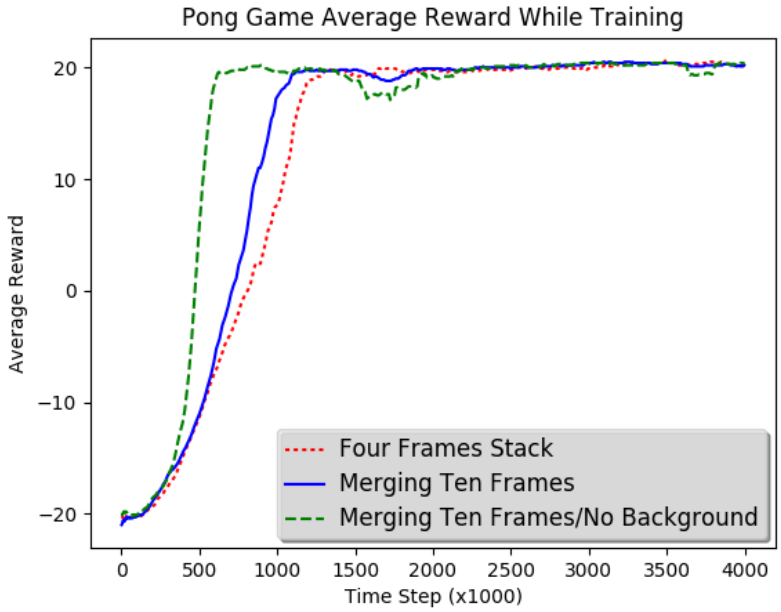


Figure 4.21: Experiment II.III - Game Average Rewards - DQN Baseline (Stack vs Merging Ten Frames vs Merging Ten Frames Without Background)

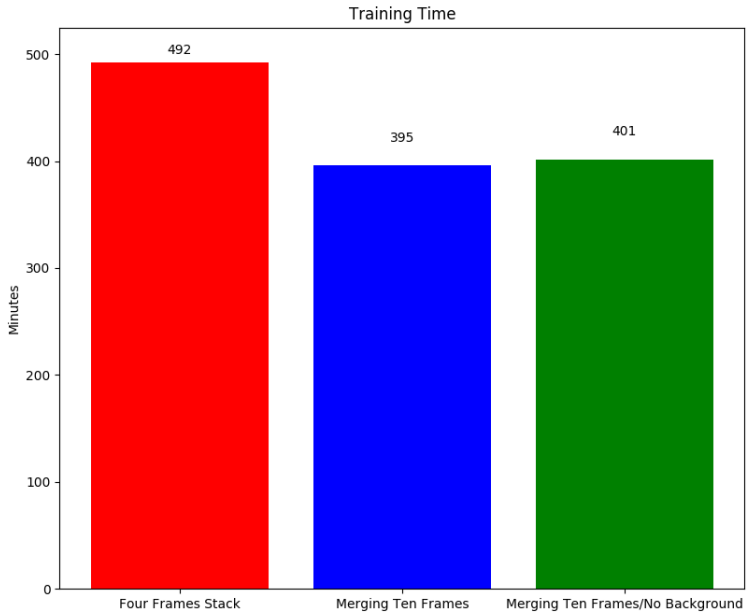


Figure 4.22: Experiment II.III - Training Time - DQN Baseline (Stack vs Merging Ten Frames vs Merging Ten Frames Without Background)

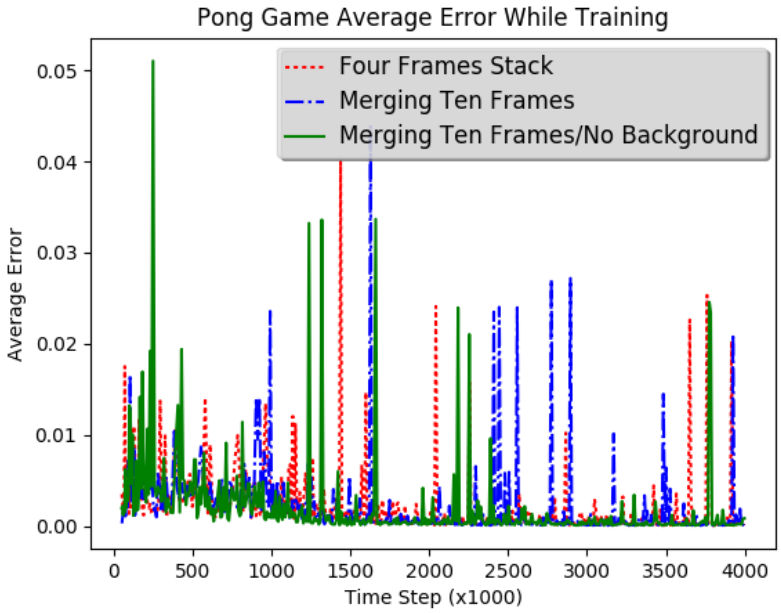


Figure 4.23: Experiment II.III - Average Errors - DQN Baseline (Stack vs Merging Ten Frames vs Merging Ten Frames Without Background)

Experiment II.IV

In this experiment, the objective was to test the effect of the merging technique on DDQN which is an improvement over DQN algorithm. DDQN is part of OpenAI baseline algorithms.

Figure 4.25 shows that by replacing the stack of four frames with one frame, the training time can be reduced by 28%. However, this comes with a price of performance degradation similar to what happened with DQN (Figure 4.24).

The experiments have shown that the merging technique can in fact improve the performance of DDQN. Figure 4.26 shows that when merging ten frames the performance can exceed slightly the stack version of DDQN while removing the background can outperform it with a good margin. All this improvement in performance comes with a reduction in training time by around 25% when merging the frames and by 23% when the merge is accompanied with background removal.

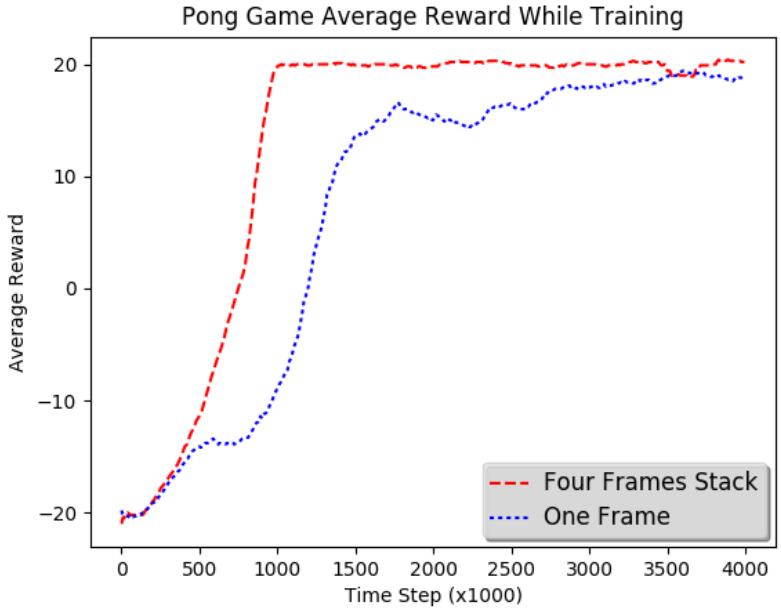


Figure 4.24: Experiment II.IV - Game Average Rewards - DDQN Baseline (Stack vs One Frame)

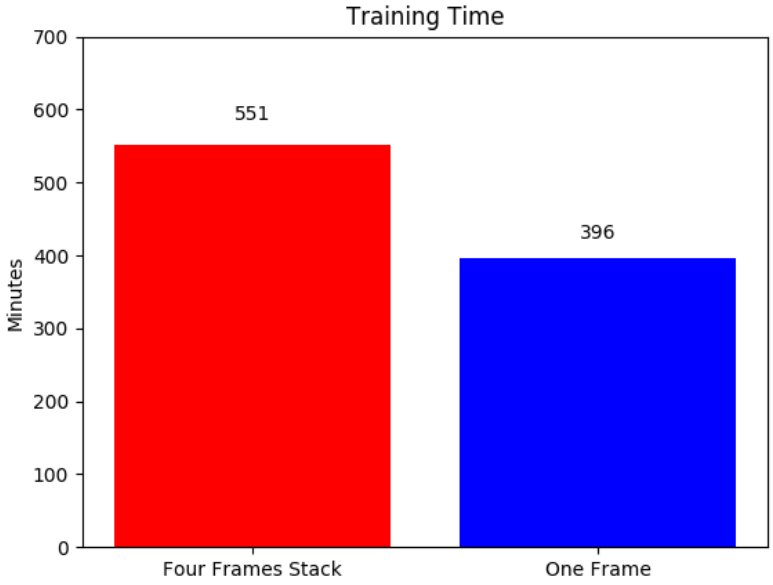


Figure 4.25: Experiment II.IV - Training Time - DDQN Baseline (Stack vs One Frame)

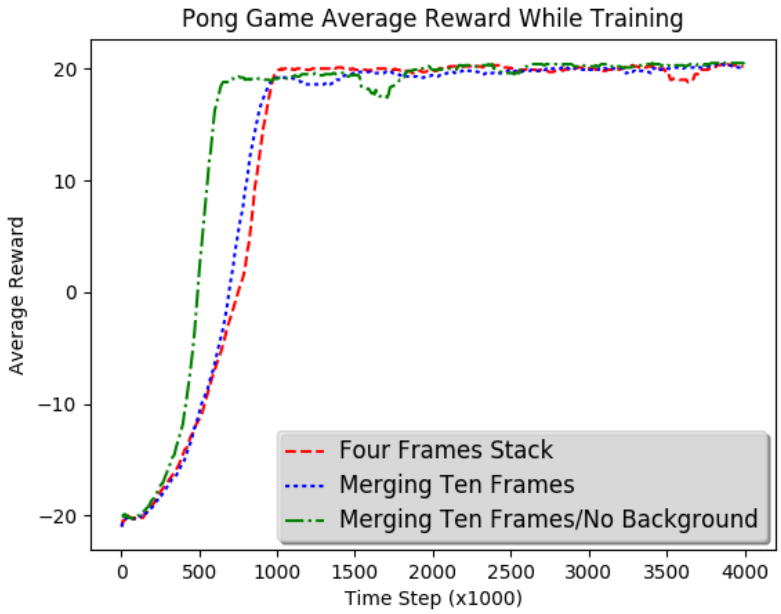


Figure 4.26: Experiment II.IV - Game Average Rewards - DDQN Baseline (Stack vs Merging Ten Frames vs Merging Ten Frames Without Background)

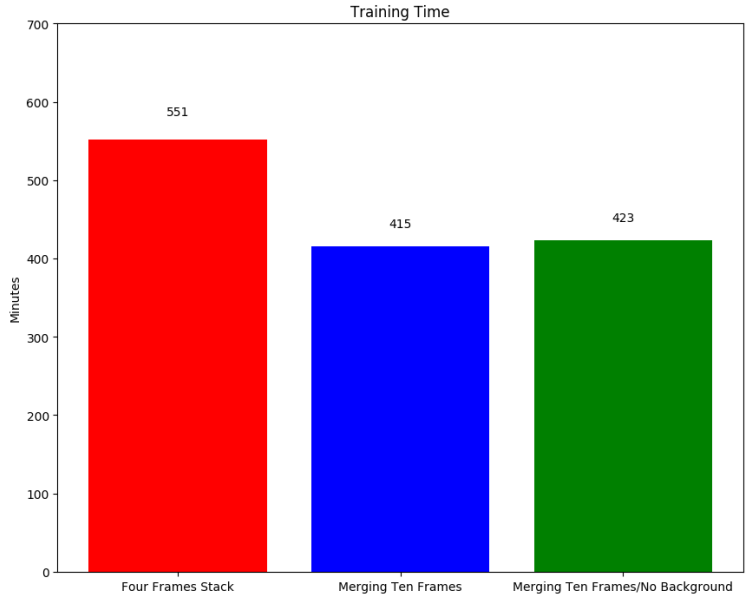


Figure 4.27: Experiment II.IV - Training Time - DDQN Baseline (Stack vs Merging Ten Frames vs Merging Ten Frames Without Background)

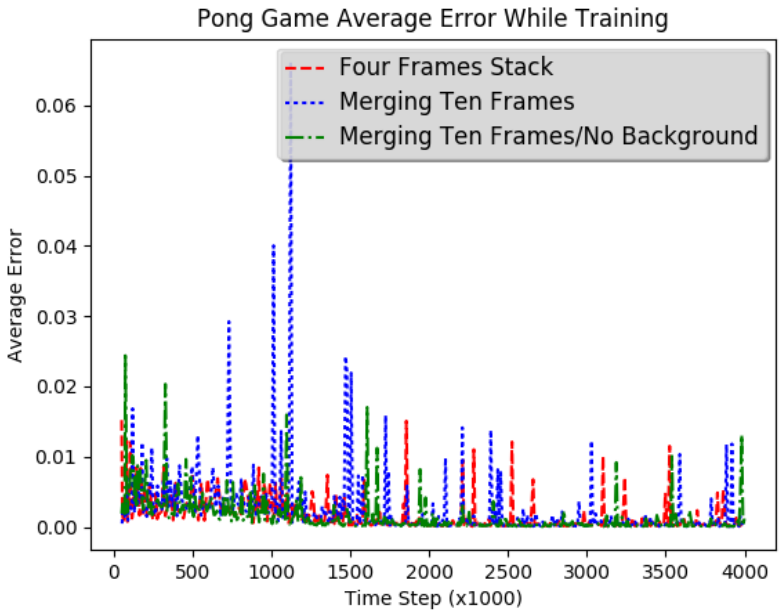


Figure 4.28: Experiment II.IV - Average Errors - DDQN Baseline (Stack vs Merging Ten Frames vs Merging Ten Frames Without Background)

Experiment II.V

In this experiment, the objective was to test the effectiveness of the merging technique on Dueling DQN especially that the architecture of the CNN has been altered in this algorithm from the DQN. Dueling DQN is also part of OpenAI baseline algorithms.

In all previous experiments, the behaviour of replacing the stack of frames with one single frame has been similar, a degradation in performance and a decrement in training time. The same behaviour happened with Dueling DQN as Figures 4.29 and 4.30 show with a reduction in training time by around 26%.

When the merging technique is applied, Dueling DQN has gained a boost in performance in both cases when using merged frames with and without background with an advantage to the latter. The reduction in training time was by around 23% for merged frames and by around 22% when the background was removed.

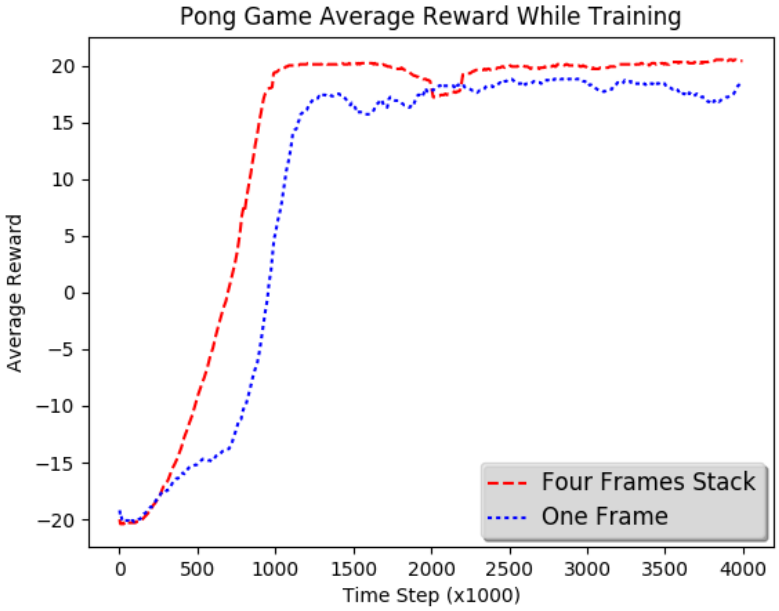


Figure 4.29: Experiment II.V - Game Average Rewards - DuelingDQN Baseline (Stack vs One Frame)

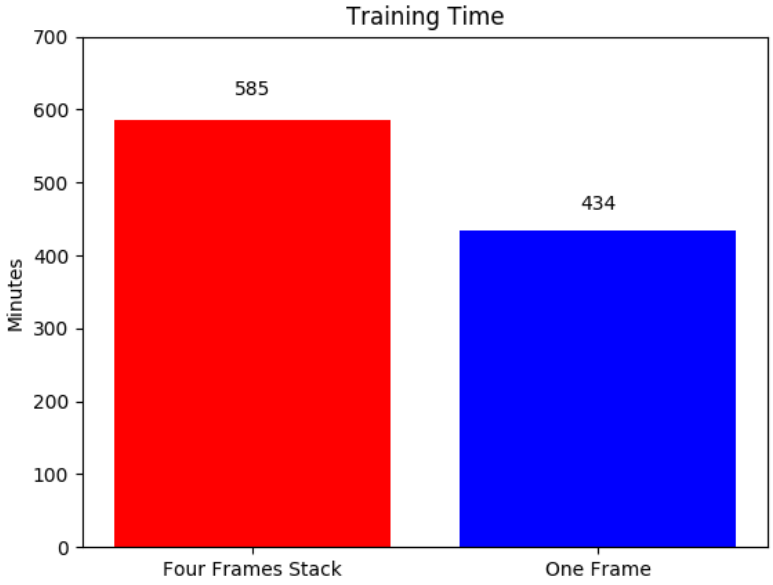


Figure 4.30: Experiment II.V - Training Time - DuelingDQN Baseline (Stack vs One Frame)

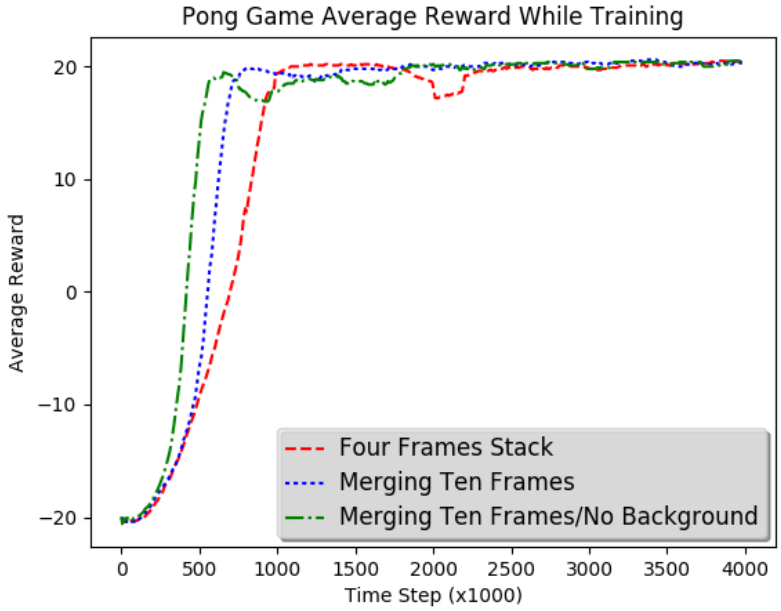


Figure 4.31: Experiment II.V - Game Average Rewards - DuelingDQN Baseline (Stack vs Merging Ten Frames vs Merging Ten Frames Without Background)

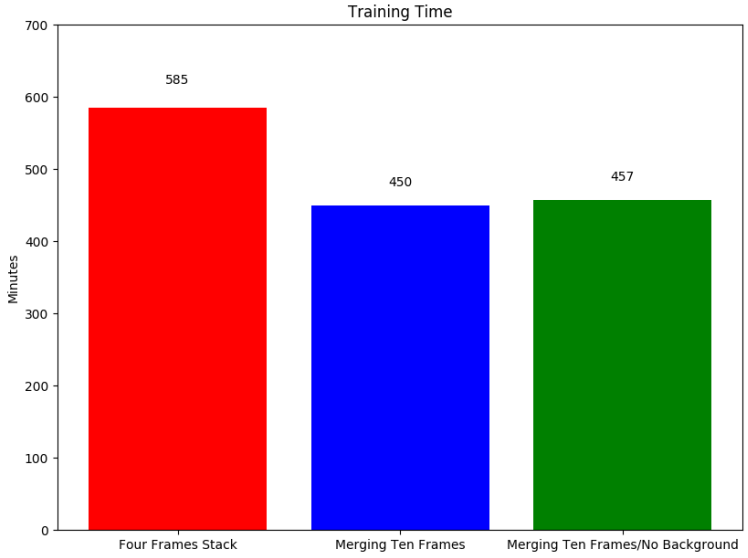


Figure 4.32: Experiment II.V - Training Time - DuelingDQN Baseline (Stack vs Merging Ten Frames vs Merging Ten Frames Without Background)

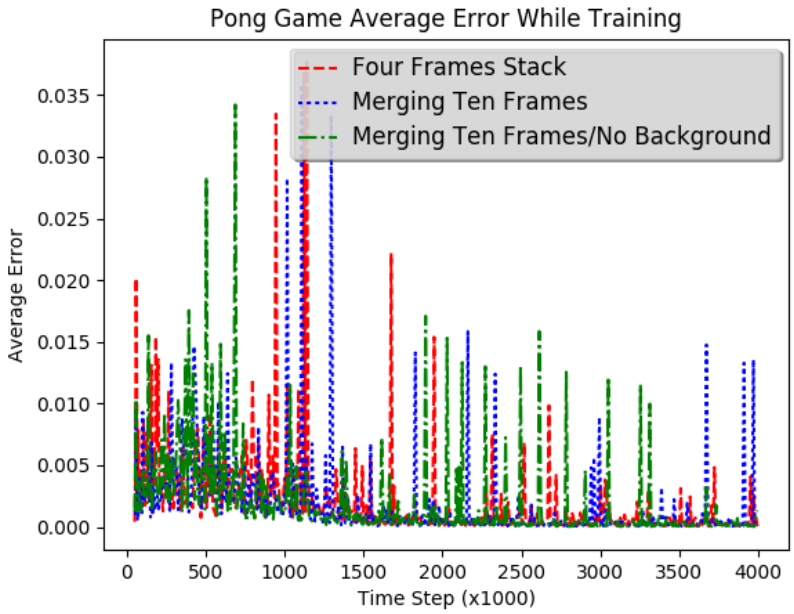


Figure 4.33: Experiment II.V - Average Errors - DuelingDQN Baseline (Stack vs Merging Ten Frames vs Merging Ten Frames Without Background)

4.3.3 Experiments III

This set of experiments was conducted to test the effect of the merging technique on DQN while being trained on complex frames as the ones presented by ViZDoom. The implementation that was used in this set of experiments is the skeleton DQN. The experiments were conducted on two scenarios from ViZDoom that were described in section 2.3.2.

Experiment III.I

In this experiment, the goal was to test the performance of DQN equipped with the merging technique against DQN with a stack of frames. They both used four frames. The test was against the basic scenario from ViZDoom. Figure 4.34 shows that their performance was very similar in this game. However, the training time required to train DQN with merging technique was less than the one with stack of frames by around 28% as can be seen in Figure 4.35. This proves again that the merging technique can offer a reduction in training time while maintaining the performance of trained agents.

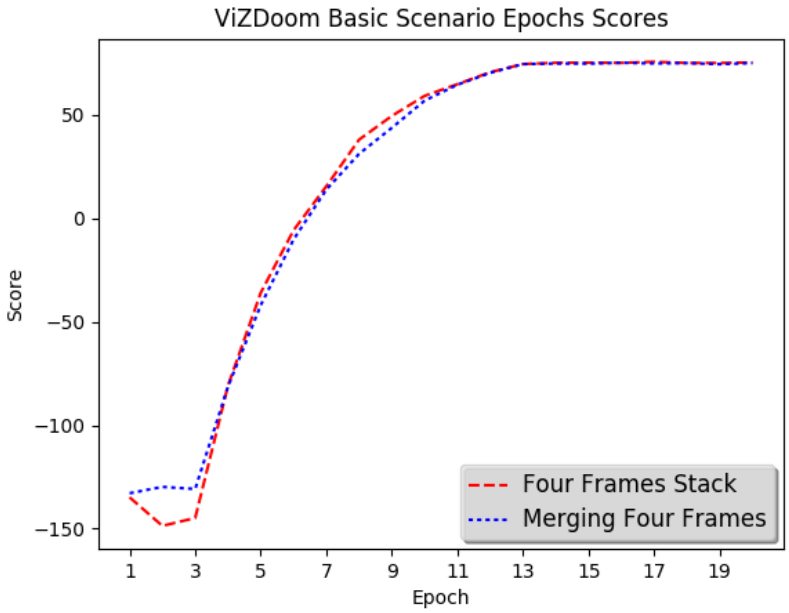


Figure 4.34: Experiment III.I - Game Average Rewards - DQN (Stack vs Merging Four Frames)

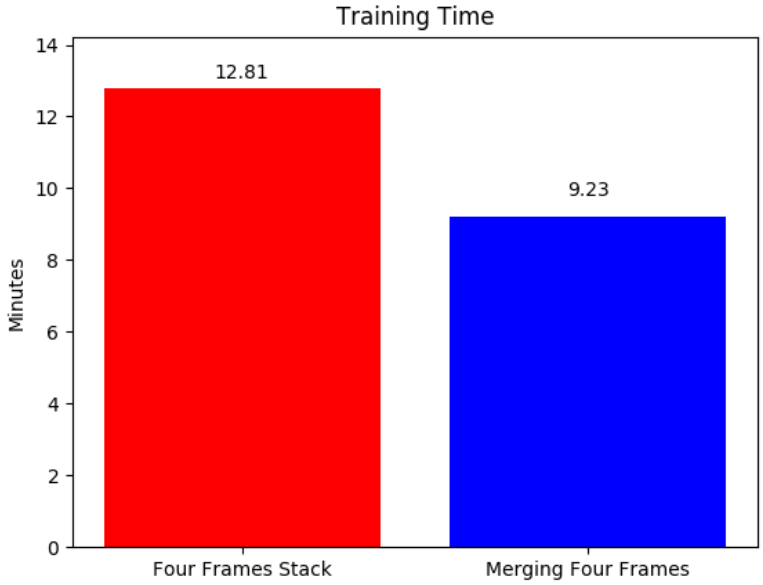


Figure 4.35: Experiment III.I - Training Time - DQN (Stack vs Merging Four Frames)

Experiment III.II

This experiment was conducted to test more complex scenario from ViZDoom and study the impact on performance when comparing the merging technique against the stack of frames in DQN. Figure 4.36 shows that in this more complex scenario, the merging technique helped the DQN agent in learning better policy that allowed it to surpass the performance of the stack version. The interesting part is that the merging technique required less training time than the stack version by around 29%.

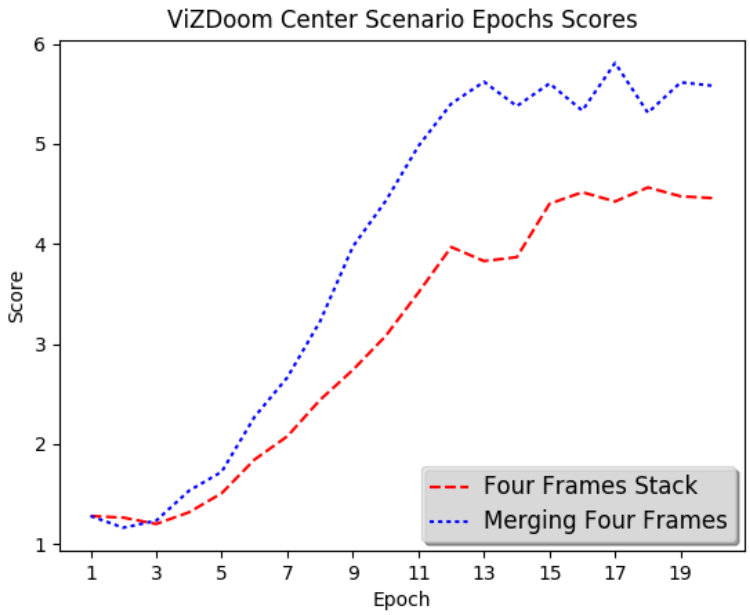


Figure 4.36: Experiment III.II - Game Average Rewards - DQN (Stack vs Merging Ten Frames)

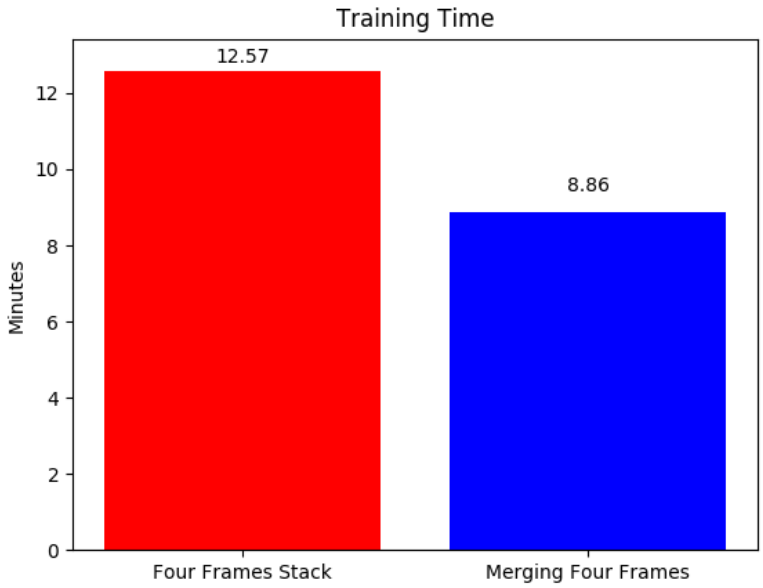


Figure 4.37: Experiment III.II - Training Time - DQN (Stack vs Merging Ten Frames)

4.3.4 Results Summary

The experiments have shown that the merging technique is in fact capable of improving DQN model in terms of training speed and performance regardless of the implementation used or the complexity of the testing environment. In fact, the results have shown that when the implementation is not fully optimized, its training speed can be improved more. This was clear when comparing the baseline DQN before and after using the merging technique. The training time has been reduced by 18% to 20% based on removing the background or keeping it. However, when it comes to the faster skeleton DQN, the reduction in training time was by 10%. Moreover, the experiments have proved that the merging technique can be effective on DDQN and Dueling DQN. Their training time has been reduced by 23% to 25% and 22% to 23% respectively based on removing the background or keeping it. In terms of performance, skeleton DQN has shown a slight improvement in performance when the merging technique was accompanied with background removal. On the other hand, all baseline models whether DQN, DDQN or Dueling DQN have shown an improvement in performance even when the background is kept. In fact, when the background was removed the performance was gaining a good boost in all models. With such technique, it is now possible to train a better version of Dueling DQN that is by itself an improvement over DQN in less time than what it would take to train a baseline DQN with a stack of frames. When the merging technique was tested on more complex environments from ViZDoom, the results have shown that it can boost the performance of DQN to surpass the stack

version or at least maintain similar performance. In both cases, the training time required by the merging technique was less than the one took the stack version by 28% to 29%.

	Training Time	
Model	Skeleton DQN	OpenAI Baselines
Pong Game		
DQN - Four Frames Stack	313	492
DQN - One Frame	261	375
DQN - Merging Four Frames	271	383
DQN - Merging Six Frames	272	389
DQN - Merging Eight Frames	273	391
DQN - Merging Ten Frames	282	395
DQN - Merging Four Frames Without Background	259	393
DQN - Merging Ten Frames Without Background	280	401
DDQN - Four Frames Stack	—	551
DDQN - One Frame	—	396
DDQN - Merging Ten Frames	—	415
DDQN - Merging Ten Frames Without Background	—	423
DuelingDQN - Four Frames Stack	—	585
DuelingDQN - One Frame	—	434
DuelingDQN - Merging Ten Frames	—	450
DuelingDQN-Merging Ten Frames w/o Background	—	457
ViZDoom - Basic Scenario		
DQN - Four Frames Stack	12.8	—
DQN - Merging Four Frames	9.23	—
ViZDoom - Defend The Center Scenario		
DQN - Four Frames Stack	12.57	—
DQN - Merging Four Frames	8.86	—

Table 4.1: Training Time of Different Models

CHAPTER 5

CONCLUSION

All the researchers in the literature are focusing on improving the performance of DQN which is considered to be the first RL algorithm that utilizes CNN as value function approximator. However, their improvements tend to increase the complexity of the developed models or at least maintain the same level of complexity as DQN. However, in this thesis, it has been suggested a new technique that can reduce the complexity of DQN which results in improving its training speed while maintaining its performance if not exceeding it. The experiments have proven that the technique is implementation independent. The technique has been tested on two different implementations of DQN. In both cases the training time was reduced however with different percentages. In the first case where the implementation was based on a skeleton DQN, the training time was reduced by 10%. On the other hand, in the implementation that was based on OpenAI baseline, the training time was reduced by 18%. However, it is important to be noted that the percentage in reduction does not reflect the actual training time since the

experiments have shown that the first implementation is in fact faster than the second one by 36%, yet in both cases the merging technique was capable in reducing the training time necessary. In terms of performance, merging the frames was not sufficient to reach the performance of stack version of DQN in the first implementation and the background needed to be removed in order to slightly outperform the stack version when it was tested on Pong. However, when tested on ViZDoom, the merging technique was sufficient to reach the performance of stack version and even exceeding it. On the other side, with baseline DQN, merging the frames was more than enough to reach the performance of stack DQN and to even exceed it while removing the background has gained baseline DQN a boost in performance over the stack version when it was tested on Pong game. Further more, the experiments have shown that the merging technique is not limited to DQN. In fact, it has been proven that even DDQN and Dueling DQN can have a boost in performance and a reduction in training time when equipped with the merging technique. In fact, it was shown that Dueling DQN with merged frames without background can outperform the stack version of DQN while finishing training by 7% less time. The merging technique has shown its capability on improving DQN regardless of the complexity of the environment and the frames it generates. This was the case when tested against simple game as Pong and when it encountered with complex frames as the ones presented by ViZDoom All this, have shown that the merging technique is promising and can help in optimizing DRL algorithms by boosting their performance while reducing the time necessary to train them.

Future Work

The proposed technique has shown a capability to be applied on other algorithms than DQN including DDQN and Dueling DQN. This in fact raises a question whether this technique can be applied to further other algorithms in DRL while having the same impact or not. This can be an interesting topic to be studied in future work. Furthermore, all the experiments that have been conducted in this thesis were applied on three games only the Pong game and two scenarios from ViZDoom. More games can be used from ALE in addition to other more complex challenges from VizDoom to test for the robustness of the proposed technique on the performance in other environments.

Threats to Validity

One of the threats to the validity of the work is that all the experiments have been conducted once. Even though the results have shown an improvement in performance and reduction in training time. However, both tested factors can be affected by the exploration strategy followed by the agent. Therefore, to ensure the validity of the experiments several runs must be taken into consideration before building confidence in the results. Furthermore, all the experiments have been conducted on one machine. Perhaps the reduced percentage in training time is bound to the machine that was used in the experiments. Therefore, to ensure that this technique can in fact reduce the training time by this amount, it must be tested in different setups. Finally, testing the technique using few games is

not enough for generalizing the findings. Perhaps more sophisticated games and environments can in fact reduce the performance of the merging technique.

REFERENCES

[1] M. Campbell, A. J. Hoane, and F.-h. Hsu, "Deep blue," *Artificial intelligence*, vol. 134, no. 1-2, pp. 57–83, 2002.

[2] S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik, and D. D. Edwards, *Artificial intelligence: a modern approach*. Prentice hall Upper Saddle River, 2003, vol. 2, no. 9.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[4] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning." in *AAAI*, 2016, pp. 2094–2100.

[5] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning*, 2016, pp. 1928–1937.

[6] D. Hassabis, D. Kumaran, C. Summerfield, and M. Botvinick, "Neuroscience-inspired artificial intelligence," *Neuron*, vol. 95, no. 2, pp. 245–258, 2017.

[7] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1, no. 1.

[8] M. Wiering and M. van Otterlo, *Reinforcement Learning: State-of-the-art*. Springer Science & Business Media, 2012, vol. 12.

[9] J. P. O'Doherty, S. W. Lee, and D. McNamee, "The structure of reinforcement-learning mechanisms in the human brain," *Current Opinion in Behavioral Sciences*, vol. 1, pp. 94–100, 2015.

[10] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-carlo tree search: A new framework for game ai." in *AIIDE*, 2008.

[11] O. Sigaud and O. Buffet, *Markov decision processes in artificial intelligence*. John Wiley & Sons, 2013.

[12] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[13] S. Kumar, *Neural networks: a classroom approach*. Tata McGraw-Hill Education, 2004.

[14] S. S. Haykin, S. S. Haykin, S. S. Haykin, and S. S. Haykin, *Neural networks and learning machines*. Pearson Upper Saddle River, NJ, USA:, 2009, vol. 3.

[15] N. Michael, "Neural networks and deep learning," 2015.

[16] P. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences. ph. d. thesis, harvard university, cambridge, ma, 1974." 1974.

[17] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, p. 533, 1986.

[18] J. Soni, U. Ansari, D. Sharma, and S. Soni, "Predictive data mining for medical diagnosis: An overview of heart disease prediction," *International Journal of Computer Applications*, vol. 17, no. 8, pp. 43–48, 2011.

[19] H. Yan, Y. Jiang, J. Zheng, C. Peng, and Q. Li, "A multilayer perceptron-based medical decision support system for heart disease diagnosis," *Expert Systems with Applications*, vol. 30, no. 2, pp. 272–281, 2006.

[20] S.-B. Cho, "Neural-network classifiers for recognizing totally unconstrained handwritten numerals," *IEEE Transactions on Neural Networks*, vol. 8, no. 1, pp. 43–53, 1997.

[21] S. Knerr, L. Personnaz, and G. Dreyfus, "Handwritten digit recognition by neural networks with single-layer training," *IEEE Transactions on neural networks*, vol. 3, no. 6, pp. 962–968, 1992.

[22] A. Ahad, A. Fayyaz, and T. Mehmood, "Speech recognition using multilayer perceptron," in *Students Conference, 2002. ISCON'02. Proceedings. IEEE*, vol. 1. IEEE, 2002, pp. 103–109.

[23] R. Feraund, O. J. Bernier, J.-E. Viallet, and M. Collobert, "A fast and accurate face detector based on neural networks," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 23, no. 1, pp. 42–53, 2001.

[24] Y.-S. Ryu and S.-Y. Oh, "Automatic extraction of eye and mouth fields from a face image using eigenfeatures and multilayer perceptrons," *Pattern recognition*, vol. 34, no. 12, pp. 2459–2466, 2001.

[25] P. Pandey and S. Barai, "Multilayer perceptron in damage detection of bridge structures," *Computers & structures*, vol. 54, no. 4, pp. 597–608, 1995.

[26] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[27] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[28] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[29] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[30] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop, coursera: Neural networks for machine learning," *University of Toronto, Technical Report*, 2012.

[31] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[32] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, 2012.

[33] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaśkowski, "Vizdoom: A doom-based ai research platform for visual reinforcement learning," *arXiv preprint arXiv:1605.02097*, 2016.

[34] A. Mahendran, H. Bilen, J. F. Henriques, and A. Vedaldi, "Researchdoom and cocodoom: Learning computer vision with games," *arXiv preprint arXiv:1610.02431*, 2016.

[35] G. Tesauro, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[36] M. Riedmiller, "Neural fitted q iteration-first experiences with a data efficient neural reinforcement learning method," in *ECML*, vol. 3720. Springer, 2005, pp. 317–328.

[37] G.-S. Yang, E.-K. Chen, and C.-W. An, "Mobile robot navigation using neural q-learning," in *Machine Learning and Cybernetics, 2004. Proceedings of 2004 International Conference on*, vol. 1. IEEE, 2004, pp. 48–52.

[38] S. Lange, M. Riedmiller, and A. Voigtlander, "Autonomous reinforcement learning on raw visual input data in a real world application," in *Neural Networks (IJCNN), The 2012 International Joint Conference on.* IEEE, 2012, pp. 1–8.

[39] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[40] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[41] S. Thrun and A. Schwartz, "Issues in using function approximation for reinforcement learning," in *Proceedings of the 1993 Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum*, 1993.

[42] H. V. Hasselt, "Double q-learning," in *Advances in Neural Information Processing Systems*, 2010, pp. 2613–2621.

[43] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling network architectures for deep reinforcement learning," *arXiv preprint arXiv:1511.06581*, 2015.

[44] M. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," *CoRR, abs/1507.06527*, 2015.

[45] G. Lample and D. S. Chaplot, "Playing fps games with deep reinforcement learning." in *AAAI*, 2017, pp. 2140–2146.

[46] S. Alvernaz and J. Togelius, "Autoencoder-augmented neuroevolution for visual doom playing," *arXiv preprint arXiv:1707.03902*, 2017.

[47] X. Guo, S. Singh, H. Lee, R. L. Lewis, and X. Wang, "Deep learning for real-time atari game play using offline monte-carlo tree search planning," in *Advances in neural information processing systems*, 2014, pp. 3338–3346.

[48] M. Lai, "Giraffe: Using deep reinforcement learning to play chess," *arXiv preprint arXiv:1509.01549*, 2015.

[49] A. Nandy and M. Biswas, "Reinforcement learning with keras, tensorflow, and chainerrl," in *Reinforcement Learning*. Springer, 2018, pp. 129–153.

[50] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[51] C. J. Maddison, A. Huang, I. Sutskever, and D. Silver, "Move evaluation in go using deep convolutional neural networks," *arXiv preprint arXiv:1412.6564*, 2014.

[52] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, "Curiosity-driven exploration by self-supervised prediction," *arXiv preprint arXiv:1705.05363*, 2017.

[53] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[54] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[55] S. Levine. (Fall, 2017) Cs 294: Deep reinforcement learning. University Course. [Online]. Available: http://rll.berkeley.edu/deeprlcourse/

[56] P. Dhariwal, C. Hesse, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Openai baselines," https://github.com/openai/baselines, 2017.

Vitae

- Name: Anas Mohammed Albaghajati
- Nationality: Syrian Nationality
- Email: *a.baghajati@gmail.com*

- Education:
  - Master of Science degree in Computer Science, King Fahd University of Petroleum & Minerals, Dhahran, Saudi Arabia, January 2018.
  - Bachelor of Science degree in Computer Engineering, King Fahd University of Petroleum & Minerals, Dhahran, Saudi Arabia, January 2014.
- Patents:
  - **A. M. Albaghajati**, M. T. Nasir, L. Ghouti, and S. El Ferik, "Seismic sensor deployment with a stereographically configured robot," February 13 2018, US Patent 9,891,626.
  - **A. M. Albaghajati**, M. T. Nasir, L. Ghouti, and S. El Ferik, "Apparatus and method for deploying sensors," October 24 2017, US Patent 9,798,327.